

# 研发反模式

Version 0.8

作者 叶孝鑫 [yexiaoxin@baidu.com](mailto:yexiaoxin@baidu.com)

王亦乐 [wangyile@baidu.com](mailto:wangyile@baidu.com)

百度 LBSQA 人民出版社

2013/8/2



目录

研发反模式.....1

    前言.....4

        ➤ 什么是反模式? .....4

        ➤ 本书有什么? .....4

        ➤ 谁应该读这本书? .....4

        ➤ 致谢.....4

        ➤ 反馈意见.....4

1. 架构反模式.....5

    ➤ 模块设计.....5

    ➤ 模块交互.....8

    ➤ 系统内部一致性.....18

    ➤ 对外依赖.....26

    ➤ 对外服务.....30

    ➤ 数据存储.....32

    ➤ 架构风险控制.....33

2. 编码反模式.....37

    ➤ 变量初始化.....37

    ➤ 函数接口.....40

    ➤ 异常处理.....45

    ➤ 基础库使用.....49

    ➤ 基础库开发.....50

    ➤ 性能相关.....50

    ➤ 编码规范.....52

3. 运维反模式.....55

    ➤ 上线过程.....55

    ➤ 线上环境.....58

    ➤ 线上监控.....63

## 前言

### ➤ 什么是反模式？

与我们所说的模式，或者正模式不同，反模式告诉你：“这么搞事情，一定会把事情搞砸”。

### ➤ 本书有什么？

本书作者在 **baidu** 多年的 QA 工作基础上，总结了研发过程中常见的错误模式，分门别类，称为研发反模式，这些反模式分成三大部分，分别是架构反模式，编码反模式和运维反模式，每个反模式都以具体的例子来讲解，这些例子，没有一个是瞎编的。作者保证：每一个例子都是发生在 **baidu** 内部真实的故事。

### ➤ 谁应该读这本书？

本书面向关心自己产品质量的 **rd**，**qa** 和 **op** 同学，作者希望本书能够成为大家日常工作的一本参考书，经常拿起来审视一下自己是否命中这些反模式，有则改之，无则加勉。

### ➤ 致谢

感想我们的经理王新能，是他提议我们写这本书，在写作过程中还不断地给我们鼓励和支持。

感谢为本书提交修改意见的同学，他们是：

李瀚(LBSRD) 韩哲(DTQA) 杨进(PSQA) 刘英杰(LBSQA)

本书的案例除了来自于作者所在产品线的项目经历，还有大量来自公司内部网的 **wiki**，向这些无名英雄表示感谢。

### ➤ 反馈意见

由于作者水平有限，本书必定错误不少，如发现错误，请联系作者：

百度 HI：叶孝鑫 ，王亦乐 

邮件地址：[yexiaoxin@baidu.com](mailto:yexiaoxin@baidu.com); [wangyile@baidu.com](mailto:wangyile@baidu.com)

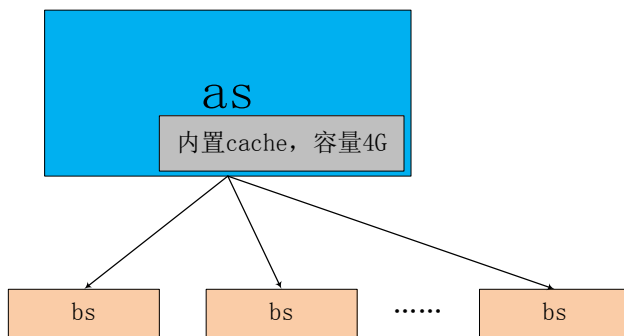
## 1. 架构反模式

### ➤ 模块设计

#### 反模式1: 自带大 cache 的模块

##### 例子

某产品线的 **as** 模块, 为降低下游 **bs** 模块的压力, 内置持久化 **cache** 功能, 耗内存 **4G**, 命中率 **85%**。此 **cache** 对线上服务至关重要, 命中率轻微的抖动都会给下游 **bs** 带来很大的压力变化。据估算, 单机房 **77** 台 **as** 机器中, 只要有 **3** 台故障, 下游就会有 **bs** 被压垮。



这带来的问题是, 上线和回滚速度慢:

由于 **as** 模块重启时候需要 **load** 持久化 **cache**, 所以重启速度比较慢, 需要 **3** 分钟时间, 另外由于担心下游 **bs** 被压垮, 所以只能两个 **as** 同时重启。这样, 替换完 **77** 个 **as** 模块, 大约需要 **2** 个小时, 上线需要 **2** 个小时, 如果是回滚, 也需要 **2** 个小时。

动不动就要 **2** 个小时的上线和回滚速度, 在系统中埋下了巨大的事故风险, **as** 的策略是经常升级的, 如果某次升级引入 **bug**, 或者模块 **core dump**, 那么我们需要 **2** 个小时的时间, 才能从错误中恢复过来。这宝贵的 **2** 个小时, 已经构成多起事故了。

##### 后果

降低系统稳定性, 随时引发事故

##### 解决方案

将 **cache** 功能从频繁升级的模块中剥离出来, 采用旁路式 **cache**, 或者 **cache** 服务化。

#### 反模式2: 自带大字典的模块

##### 例子

某产品线的 **da** 模块, 启动时需要 **load** 大约 **40G** 的字典, **da** 策略常升级, 字典功能相关代码不常升级。



## 后果

同自带大 cache 的模块

## 解决方案

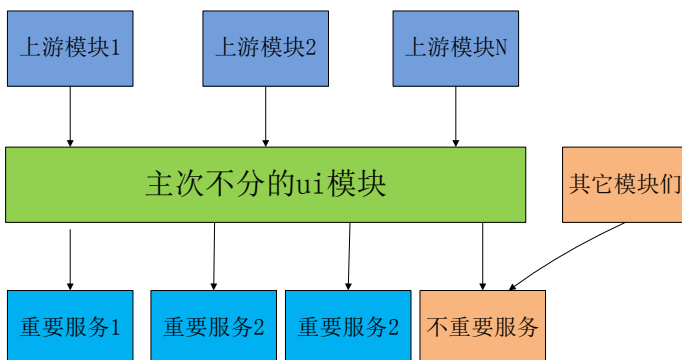
将字典功能剥离，字典服务化

## 反模式3: 主次不分的模块

### 例子

#### 例 1

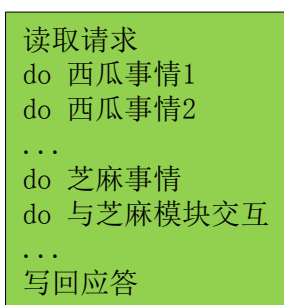
如下图，某产品线的 **ui** 模块，处于上下游的必经之路上，它集成了下游各种服务，包括重要的和不重要的，平均耗时长和平均耗时短的。



如果不重要的服务出现故障，或者由于其它模块们不厚道，把不重要的服务搞垮了，导致不重要的服务出现长耗时，那么长耗时会 **hang** 住 **ui** 模块，将长耗时传递到整个系统，将系统中的一个小的故障，放大成一个大的事故。

#### 例 2: 西瓜与芝麻的故事

这是另外一个 **ui** 模块，西瓜事情是主要业务逻辑，芝麻事情是对用户请求信息做些处理，与芝麻模块交互是为了将芝麻事情的输出，通过网络写到统一的一台机器上去，做统计，分析，改进用户体验这些事情。



可以看到，芝麻模块干的事情，其实可以是离线的，稳定性要求很低的，而这个 **ui** 模块是在线模块，稳定性要求是很高的，一旦芝麻模块稳定性出现问题，会连累 **ui** 模块的在线服务。

## 后果

降低系统的稳定性，整个系统的稳定性低于最不稳定的模块的稳定性

## 解决方案

例 1: **ui** 模块对不同下游区分对待，做垂直化拆分或者改成异步交互模式。

例 2: **ui** 模块把芝麻事情和与芝麻模块交互这些事情丢到一个专门的线程，异步处理。

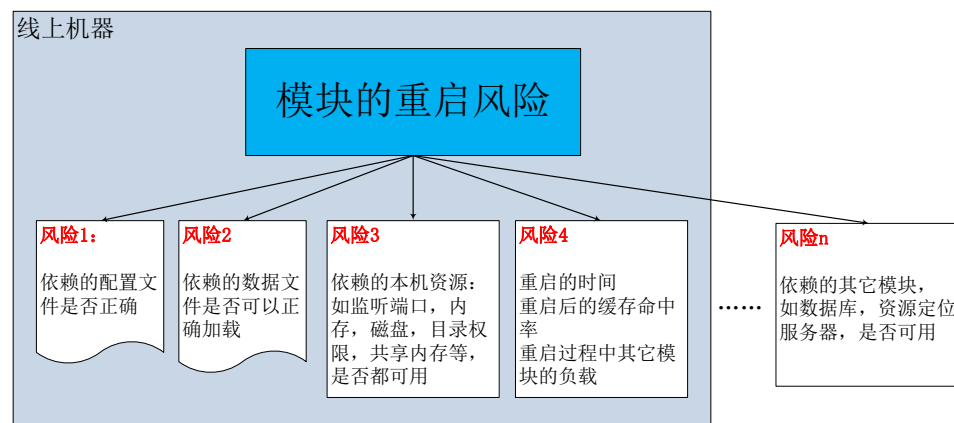
## 反模式4: 定时重启的模块

### 例子

某产品线 **bs** 模块存在内存泄露，采取定时重启 **bs** 的方式解决。

但是重启是有风险的，比如

- 1) 模块重启时间过长
- 2) 模块重启后，缓存需要预热，可能影响性能
- 3) 一般模块的重启都是暴力重启，直接 **kill** 掉，容易影响正在处理的请求
- 4) 最严重的是，如果模块能否成功重启依赖于外部条件，那么重启风险就很难控制了，这些外部条件包括但不限于：
  - a) 后端模块或者后端数据库能否连上
  - b) 能否从其它模块获取正确的信息，如资源定位信息
  - c) 需要加载的词表或者配置是否有错误
  - d) 数据文件是否过大
  - e) 共享内存是否有权限
  - f) 本机上某些目录是否存在（因为目录变更，权限变更）
  - g) 监听端口能否重新打开



某一次定时重启之前，建库端生成的 **bs** 索引文件错误，一群人眼睁睁地看着 **bs** 重启失败，一个，两个，三个……造成事故。

如果没有定时重启，bs 是通过热加载(reload)的方式更新索引的，那么这个故事的结果就不是事故，而是热加载失败报警，然后 op 同学去回滚索引。

后果

重启有风险，定时重启会将重启风险放大，放大，再放大，降低系统稳定性

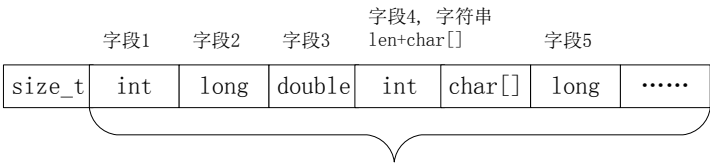
解决方案

一般情况下，需要消灭定时重启的模块

反模式5：自定义接口的模块

例子

某模块的对外接口定义为如下



读接口的代码这么写：

```
if (get_int_from_req(req, ...) != 0) { ... }
if (get_long_from_req(req, ...) != 0) { ... }
if (get_double_from_req(req, ...) != 0) { ... }
if (get_string_from_req(req, ...) != 0) { ... }
if (get_long_from_req(req, ...) != 0) { ... }
...
```

不光类型不正确读不出来，顺序错了也不行，加减字段也不方便，测试还需要专门再写一个工具，上下游模块通过这个自定义的接口严重耦合在一起。

后果

降低可维护性，可测性

解决方案

采用统一的接口，mcpack/json/protobuf 等，可维护性比所谓的效率更重要

➤ 模块交互

反模式6：使用分布不均匀的 key 当作均衡算法的输入

例子



#### 例 1

```
// query 的分布不能保证均匀，  
// 如 image 的 query='人体艺术'显然比普通 query 访问量大很多  
// 又如，某些产品线的 query='' (即空字符串)特别多  
server_index = hash(query)%server_num  
// talk to server[server_index]
```

#### 例 2

```
// baiduid 是一个 cookie，不能保证分布均匀  
// 从某些端，如 wap 过来的请求，不一定包含 baiduid(为空或者一个固定值)  
server_index = hash(baiduid)%server_num  
// talk to server[server_index]
```

### 后果

压力不均衡压垮机器，线上报警，影响服务

### 解决方案

做好调研，确保均衡算法出来的结果是比较均衡的

## 反模式7：固定的重试序列

### 例子

#### 例 1

```
// 重试序列为：“下一台”  
server_index = hash(query)%server_num  
  
// 如果交互失败，重试下一台  
if (talk to server[server_index] failed) {  
    server_index = (server_index+1)% server_num;  
    talk to server[server_index];  
}
```

#### 例 2

```
// 重试序列为：hash 算法序列  
server_index = hash_1(query) % server_num  
  
// 如果交互失败，重试下一台  
if (talk to server[server_index] failed) {  
    server_index = hash_2(query) % server_num;  
    talk to server[server_index];  
}
```

### 后果

一个是雪崩，一个是上线时损失流量。

雪崩：假设某一类 query 的重试序列为 A→B，当 A 出现故障时，B 要承受两倍的压力，如果 B 扛不住，那么 B 的下一个也会被压垮……

上线损失流量：假设重试次数是 2，上线时如果 A 和 B 同时重启，那么重试序列为 A→B 的 query 必然无结果，也就是说上线时同时重启的机器个数不可以超过重试次数，这可能会延长上线和回滚的时间，降低服务稳定性。

## 解决方案

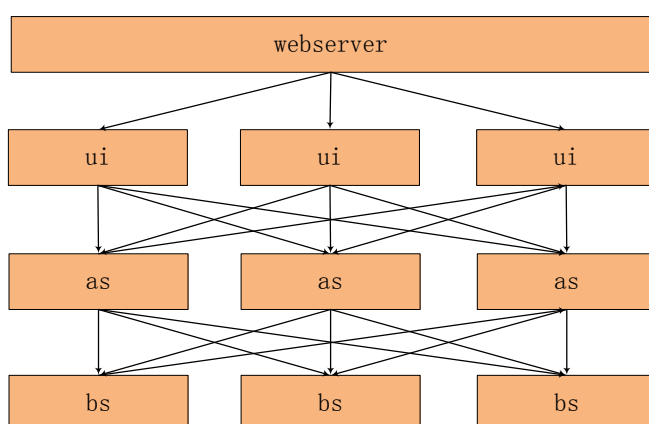
评估重试算法。

相对于固定的重试序列，随机重试序列也可能给系统带来风险，例如可能会降低下游模块的 cache 命中率，降低系统性能，甚至引起雪崩。

## 反模式8：疯狂的重试

### 例子

如图为某产品线的架构，整个系统中，上游模块对下游模块所有的交互，**重试次数都是设成 3 次**，交互失败包括连接失败，写失败，读失败这三种情形。如果是写和读失败，那么要关闭当前连接，再重新发起连接。



考虑这么一种情况，一台 bs 假死。注意是假死，不是真死，假死会导致上游模块严重超时，连接超时，写超时和读超时都有可能，真死情况下，上游模块会很快返回。

as 有 1/3 的概率需要重试，as 重试的过程中，ui 可能早就认为 as 已经超时了，所以 ui 也开始重试，ui 重试的过程中，webserver 可能认为 ui 已经超时了，所以 webserver 也开始重试……就这样，整个系统的负载急剧增加，直到系统崩溃为止。

### 后果

降低系统稳定性，一个局部的故障，引发大面积的系统故障

## 解决方案

评估重试机制

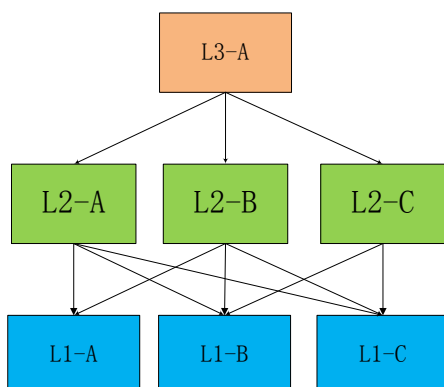
- 1) 真的需要在每一层都努力重试吗？
- 2) 真的需要这么多次重试吗？

3) 真的需要在连接，写，读这三者失败后都重试吗？

## 反模式9：不合理的超时值

### 例子

如图，某产品线的架构分三层，L3 调用 L2 的 ABC 服务，L2 调用 L1 的 ABC 服务。这个系统中，所有的超时，即 `connect_timeout`，`read_timeout` 和 `write_timeout` 全部设置为 1 秒。



考虑 L2-A 模块，最坏情况下，它与下层的交互需要  $3 \times 3 = 9$  秒，而这个时候 L3-A 早已经超时返回甚至还发起重试了，而 L2-A 的线程还在吭哧吭哧地做着无用功。

这样的超时设置下，下层只要一个模块有长耗时，就会立刻将长耗时传递到上层的所有模块，引发大面积的服务异常。

不合理的超时值，危险性极大，极易引发事故。

### 后果

降低系统稳定性，系统层数越多，交互越复杂，稳定性越差

### 解决方案

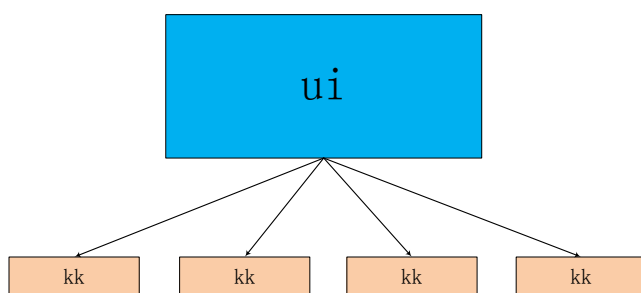
从系统整体考虑，从上层到下层分配超时，上层分配大一点，下层小一点，并且要考虑重试和本模块计算时间的影响。

## 反模式10:过度的封禁策略

### 例子

如图，线上 ui 连接 4 个 kk 模块，他们之间有这样的健康检查机制：如果在 1 分钟内，ui 与 kk 连续交互失败 5 次，则封禁这个 kk 模块 1 分钟。

但是这个封禁策略在线上的运行环境中很可能导致问题：某一次，机房内链路抖动，导致 ui 与 4 个 kk 中的 3 个连续交互失败 5 次，ui 封禁了这 3 个 kk，最后一个 kk 承受不住压力，也被压垮。



## 后果

降低服务稳定性

## 解决方案

封禁策略下，还需要有封禁保护策略，避免过封禁

## 反模式11:不考虑短连接产生的 TIME\_WAIT 状态

### 例子

线上 ui 模块，与下游 kk 模块交互，当压力大于 500 时，出现大量交互失败，ui 打印日志：

“connect: Cannot assign requested address”

原因是：短连接导致 ui 模块所在机器上存在太多 TIME\_WAIT 状态的 socket，这些 socket 占用了太多端口，后续新的 socket 无法获得新的端口，所以无法建立连接。

一个 TCP 连接关闭后，主动关闭的一方会进入 TIME\_WAIT 状态，进入这个状态的 socket 会在系统中保留 2MSL 的时间，默认 120 秒。

处于 TIME\_WAIT 状态的 socket 是会占用系统资源的，我们知道，系统用一个 4 元组来表示一个 TCP 连接

`<local addr, local port, remote addr, remote port>`

考虑 ui 机器上连接 kk 模块的 socket，他们的 local addr, remote addr, remote port 这三者都是一样的，所以连接 kk 模块的 socket 总数由 local port 的范围决定，而 local port 的范围是由系统文件 /proc/sys/net/ipv4/ip\_local\_port\_range 指定的，我们的机器上一般是 [10000, 61000]，即大约是 51000 个 local port。

上面的例子中，是 ui 主动关闭连接，所以 TIME\_WAIT 状态存在于 ui 机器，当压力达到 500，并且持续 120 秒，ui 机器上 TIME\_WAIT 状态的 socket 数为  $500 \times 120 = 60000$ ，已经超过 51000 了，所以这个时候，新的 socket 无法获得新的端口，出现 ui 连接 kk 大量失败的情形。

## 后果

最大压力受到限制，超过这个限制后，服务不稳定

## 解决方案

1. （推荐）打开系统对 TIME\_WAIT 状态的快速释放和回收选项，需要 root 权限

```
# echo 1 > /proc/sys/net/ipv4/tcp_tw_reuse
# echo 1 > /proc/sys/net/ipv4/tcp_tw_recycle
```
2. 增大 local port 的可用区间，需要 root 权限

```
# echo "1024 61000" > /proc/sys/net/ipv4/ip_local_port_range
```
3. （不推荐）禁用 TIME\_WAIT 状态：为 socket 设置 SO\_LINGER 选项

## 反模式12:禁用 TIME\_WAIT 状态

### 例子:

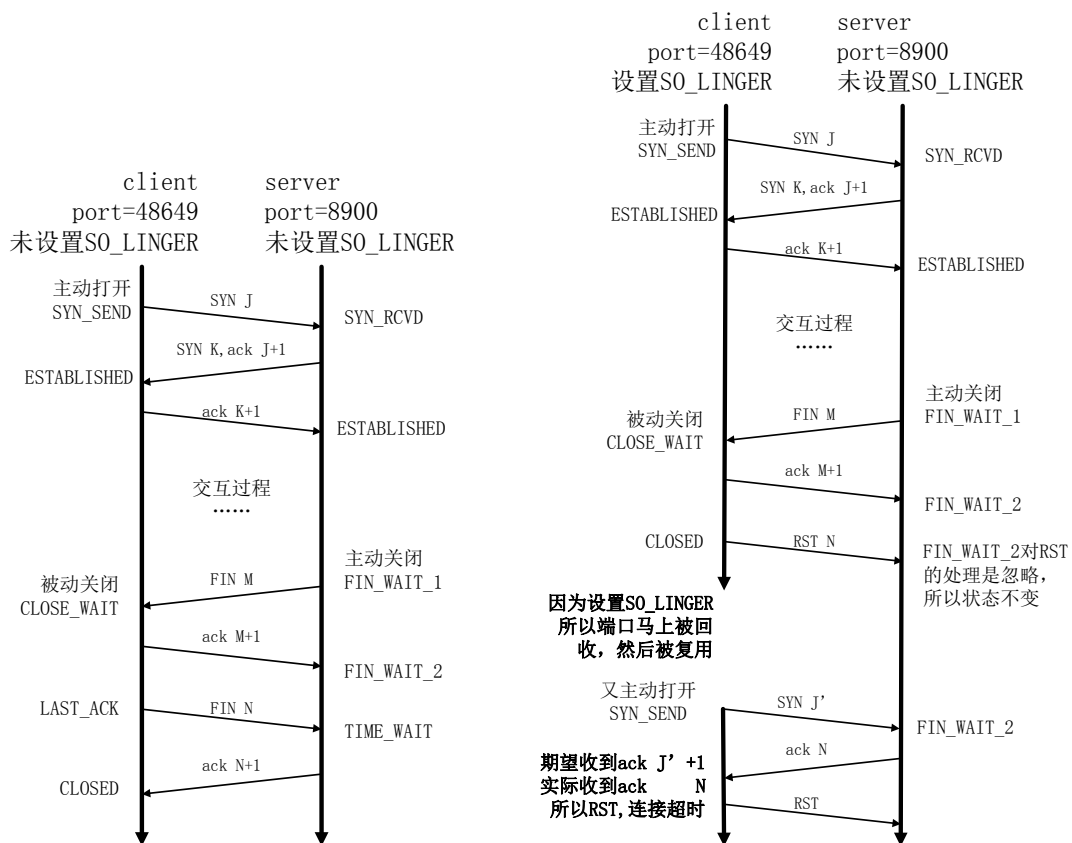
在程序中，使用 SO\_LINGER 选项可以禁用 TIME\_WAIT 状态，但是这个做法容易引发问题，需要谨慎评估。

例 1: 当 SO\_LINGER 遇上被动关闭。

如下代码，在设置了 SO\_LINGER 选项后，socket 在 close 后，会直接发送 RST(即 reset)，然后进入 CLOSED 状态，这样，就可以避免主动关闭的一方进入 TIME\_WAIT 状态。

```
struct linger ling;
ling.l_onoff = 1;
ling.l_linger = 0;
setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &ling, sizeof(ling));
```

如下图，左图是 client 和 server 都未设置 SO\_LINGER 的情况，右图是 client 设置 SO\_LINGER 却遇上 server 主动关闭的情况，悲剧了，下一个复用同一端口号的连接必定超时。



## 后果

引发连接超时，降低服务稳定性

## 解决方案

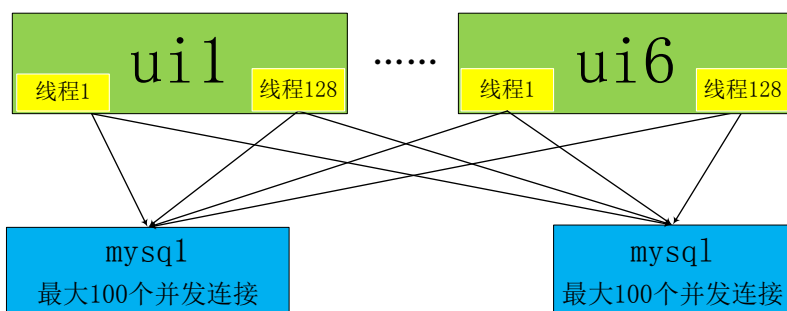
避免禁用 TIME\_WAIT 状态

## 反模式13:不评估长连接的句柄限制

### 例子

如下图，ui 与 mysql 保持长连接，6 个 ui 模块，每个包含 128 个线程，两个 mysql 部署，最大并发连接数都是 100。

那么，ui 对 mysql 保持的最大长连接数是  $6 \times 128 = 768$ ，而 mysql 的最大并发连接数只有  $2 \times 100 = 200$ ，所以，有  $768 - 200 = 568$  个 ui 线程注定要饿死，从而导致线上出现大量用户请求被拒绝。



## 后果

影响线上服务

## 解决方案

一般说来，长连接模型需要评估：

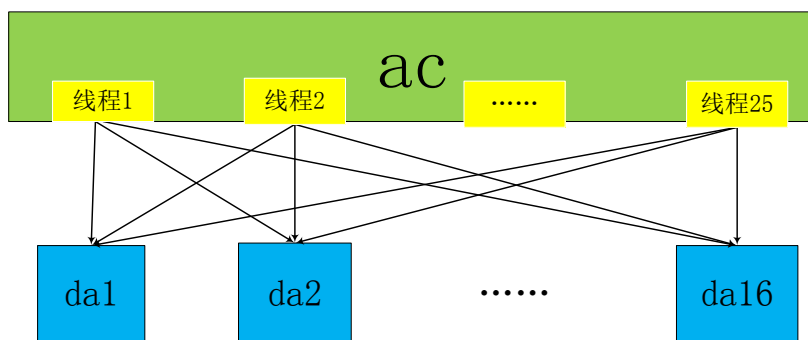
1. 上游模块句柄数=上游模块线程数×下游模块个数
2. 下游模块句柄数=上游模块线程数×上游模块个数
3. 下游模块的连接池配置：最大并发连接数，**keepalive** 时间（即多久没活动就关闭）
4. 操作系统对模块（进程）的句柄数限制，默认是 **1024**，可以通过 **ulimit** 命令修改

保持大量长连接的模块是难以横向扩展的，每次扩容或者修改线程数，都需要评估以上数值。

## 反模式14:不考虑长连接失效的情形

### 例子

如下图，ac 与 da 相连，一共 10 个 ac，每个 ac 开 25 个线程，分别与 16 个 da 保持长连接，重试策略为随机选择另外一台 da，重试 1 次。如果与某台 da 交互失败，那么要等下次与该 da 交互时才重新建立长连接。



考虑以下两种情况：

1. da 上线，需要全部重启：这个时候  $10 \times 25 \times 16 = 4000$  个长连接全部失效
2. ac 与 da 之间长时间无交互的情况，比如为了上线，整个机房被切走流量，过了一段时间（超过 **tcp keepalive** 的时间，如 2 小时）再切流量回来，这些长连接就已经全部失效了。

4000 个长连接全部失效后，无论采取何种重试策略，必定损失 PV。

## 后果

损失 PV

## 解决方案

ac 重试策略中考虑长连接失效的情形，对于长连接失效的情形（**read/write** 返回 **ECONNECTRESET**）马上重新建立连接而不是等到下一次请求再建立连接。

或者还有两个运维代价比较高的方法：

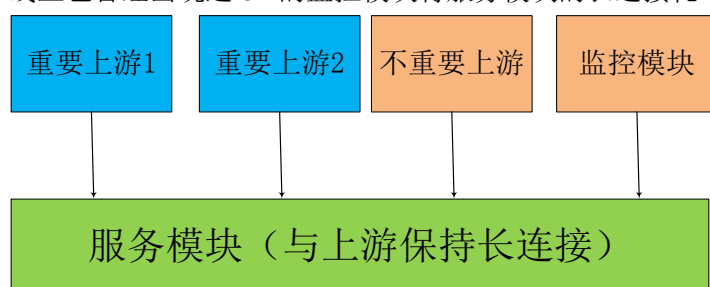
da 上线过程中，保持一定时间间隔后再重启下一台；切流量回来之前，统一重启一下 ac。

## 反模式15:不区分长连接上游模块的重要性

### 例子

如图，服务模块与上游模块保持长连接，这些上游模块中，既有重要的，又有不重要的。不重要的上游模块，相应的代码质量也会低一些，如果不重要的上游模块代码中出现 bug，有句柄泄露，那么可能会将服务模块的长连接耗尽，从而影响重要上游模块。

线上也曾经出现过 OP 的监控模块将服务模块的长连接耗尽的情况。



### 后果

降低重要上游模块的稳定性

### 解决方案

部署上区分重要上游和非重要上游

## 反模式16:不考虑并发度对网络 IO 模型的影响

### 例子

某模块线程数 50，平均处理时间是 20ms，模块的吞吐率为（理论值）大约为 2500，计算过程如下： $50 * (1000ms / 20ms) = 2500$

线下做性能测试，压力也差不多能到 2000，但是，上线后，当模块的压力值达到 700 的时候，上游模块却经常出现连接超时！

这是因为线上的并发度比线下大很多！当并发度远大于线程数 50 的时候，虽然吞吐率没有问题，但是上游模块还是有可能出现连接超时。

### 后果

降低服务稳定性

### 解决方案

并发度对网络 IO 模型的选择，以及参数设定有很大影响，需要谨慎评估和测试。

百度内部常用的 IO 模型包括 xpool, cpool, epool, apool，一般情况下，选择网络 IO 模型需要避免这两点：

- 1) 高并发场景选用 xpool

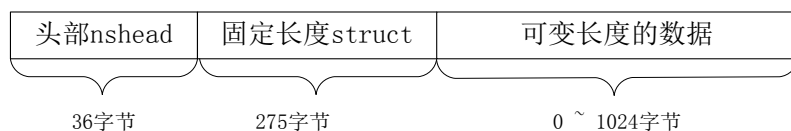


## 2) 长连接场景选用 cpool

### 反模式17:不考虑连续发送多个小数据包造成的延迟

#### 例子

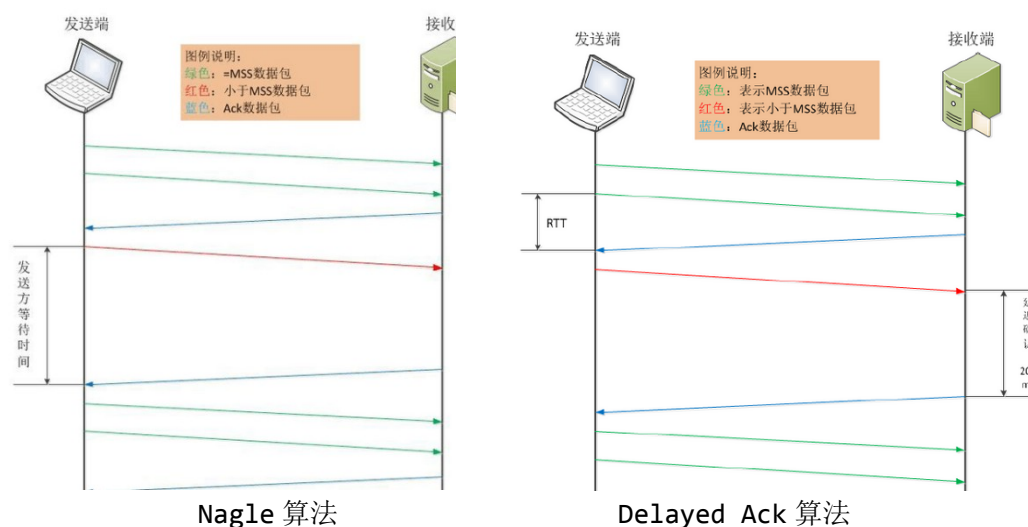
某模块与下游模块的交互接口如图所示：



在发送数据的时候，这么发送：

```
send(头部 36 字节);  
send(固定长度 struct 275 字节);  
send(可变长度 xx 字节)
```

但是这种发送方式遭遇了 Nagle 算法和 Delayed Ack 算法（如下图），导致经常出现请求处理时间无缘无故多出了 40ms，或者另外一个固定的时间值。



所以，连续多次发送小数据包的时候，需要考虑为 Nagle 算法和 Delayed Ack 算法设置正确的参数，相关的 TCP 选项包括：TCP\_NODELAY, TCP\_CORK 和 TCP\_QUICKACK。

#### 后果

导致某些请求的处理耗时增加，降低模块性能和稳定性

#### 解决方案

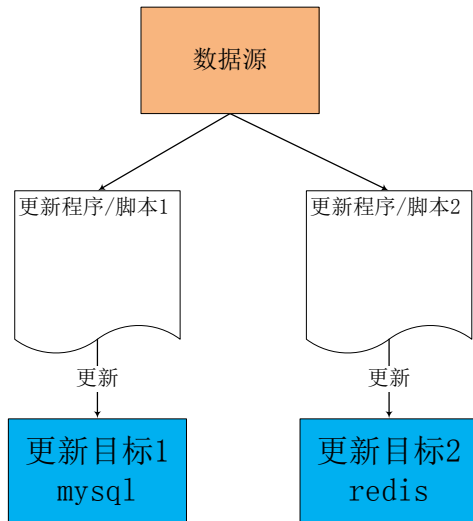
合并小数据包，或者设置正确的 TCP 参数。

## ➤ 系统内部一致性

### 反模式18:忽视容错处理导致一致性问题

#### 例子

下图中，更新程序的容错处理非常关键，一不小心，就会导致数据源和更新目标，或者更新目标 1 和更新目标 2 之间的不一致问题。



考虑以下这些情况的容错处理，如果容错处理做的不好，很容易导致两个更新目标不一致。

- 1) “陌生”的数据：因为数据源的更新频率远大于程序的更新频率，所以难保什么时候有新的格式的陌生数据进来
- 2) “奇葩”的数据：例如某些字段长度特别长，多了一些字段，少了一些字段，字段之间出现矛盾等
- 3) 数据处理过程失败：要区分多种情况，比如是请求第三方服务失败，还可以区分是网络交互失败，还是第三方服务处理失败；比如是处理过程中失败，还可以区分内部缓冲区溢出，还是计算过程溢出等，根据不同的情况，采取不同的恢复措施
- 4) 写更新目标失败：还是要小心区分多种情况，是网络交互，还是更新目标写不进去等
- 5) 如果是一个更新程序更新多个目标，需要考虑部分更新成功，部分更新失败的情况
- 6) 考虑更新程序运行到一半正 **high** 时，被 **kill** 掉的情况，重新开始会不会有副作用，会不会太慢，还是可以支持断点续传

这些情况下的容错处理，包括以下，需要根据不同情况，小心选择

- 1) 直接跳过
- 2) 记录并且跳过（根据记录，能够手工修复）
- 3) 暂停然后报警

无论如何，都不允许出现没有被处理的异常情况。

## 后果

数据不一致

## 解决方案

对整个处理流程中的所有异常做容错处理。

可以考虑更新目标之间做一致性监控，也可以考虑监控两个更新程序输出的统计信息（比如一共更新多少数据，增加多少，删除多少）

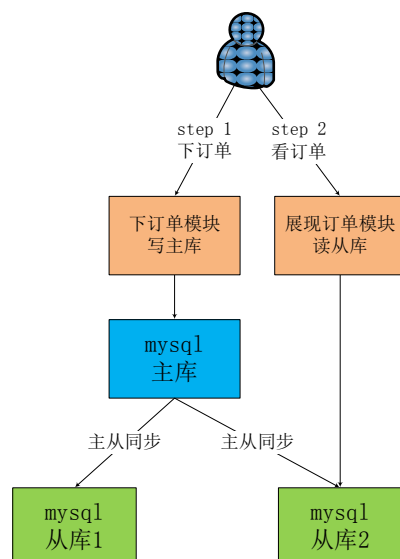
## 反模式19:忽视运维导致的一致性问题的例子

### 例子

例 1: mysql 主从同步延时

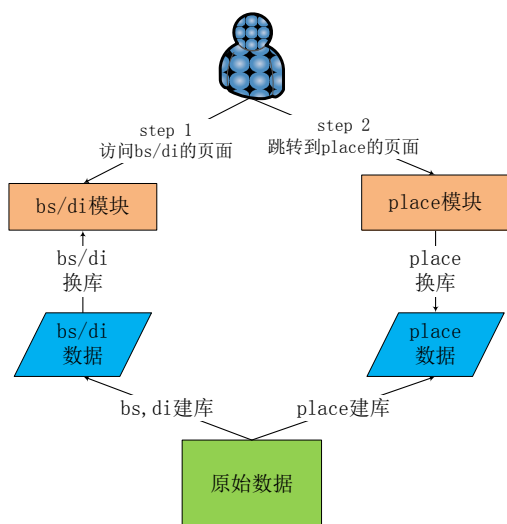
如下图，正常情况下，mysql 主从延时很短，用户下完订单之后跳转到个人中心能够看到订单，但是在真实的环境中，主从延时可能很长，比如因为网络抖动，主库突然有大量更新，或者某些从库机器负载高等原因，主从延时会达到 500 秒以上，这个时候，会出现大量用户下完订单，却看不到订单的情形。

另外，不同从库的同步延时可能还不一样，这会导致展现结果不稳定。



例 2

如下图，用户对 step1 和 step2 看到的数据是有很强的一致性要求的，但是在运维上，这一点却很难保证。



首先，二者建库时间长短不一致；其次，换库时间长短也不一致；其三，换库频率也不一定一致；其四，可能出现前者换库成功，后者换库不成功的情况；最后，维护这两套建库换库流程的可能还是两拨人！

### 后果

展现给用户不一致的数据，影响用户体验

### 解决方案

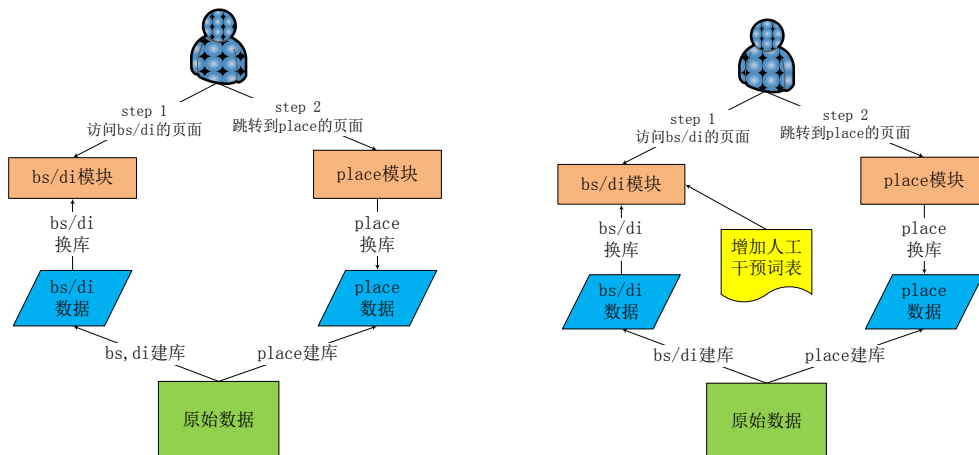
设计时候考虑运维上遇到的各种困难，技术上无法解决的，需要从流程和运维方面考虑去保证。

## 反模式20:忽视业务升级导致的一致性问题

### 例子

左图，假设原来 **step 1** 和 **step 2** 可以一致；

右图，业务升级，在 **step1** 的路径上增加人工干预，而 **step2** 的路径没有人工干预，这样，被干预的数据就不可能展现一致了；



### 后果

展现给用户不一致的数据，影响用户体验

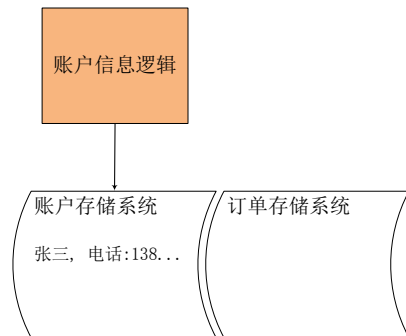
## 解决方案

增加对业务和架构的理解

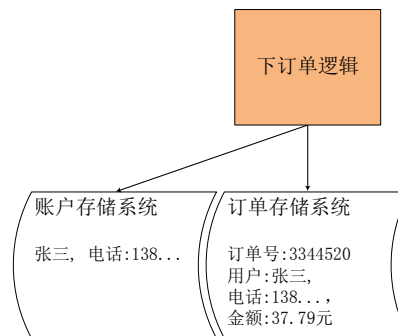
## 反模式21:忽视不同业务逻辑耦合导致的一致性问题

### 例子

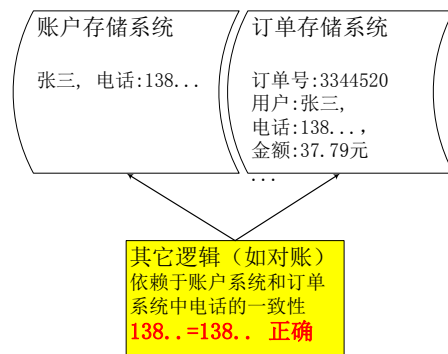
如下图，(1)(2)(3)按顺序发生，一切 OK。



图(1)

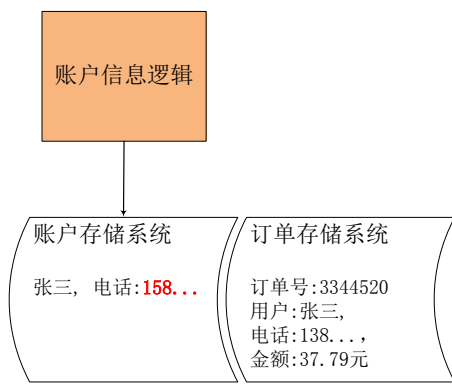


图(2)

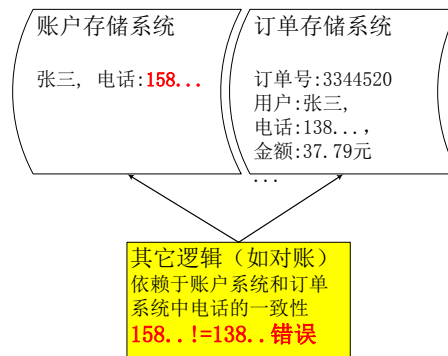


图(3)

如果在(3)之前，用户修改了自己的账户信息，那么，最后一步的逻辑就失败了！



图(3-1)



图(3-2)

## 后果

功能失效

## 解决方案

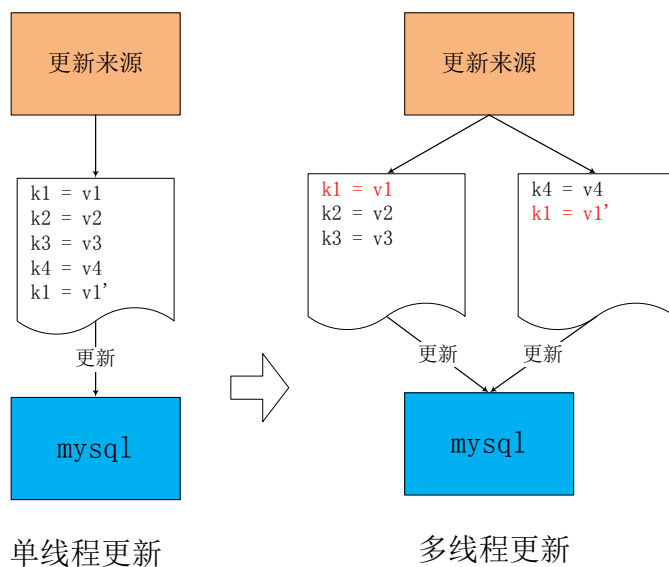
增加对业务和架构的理解

## 反模式22:忽视多任务导致的一致性问题的

### 例子

#### 例 1

如下图，为了提升效率，改成多线程，或者多进程更新之后，最终 **k1** 的值是 **v1** 还是 **v1'** 不确定的，取决于两个线程更新的速度。

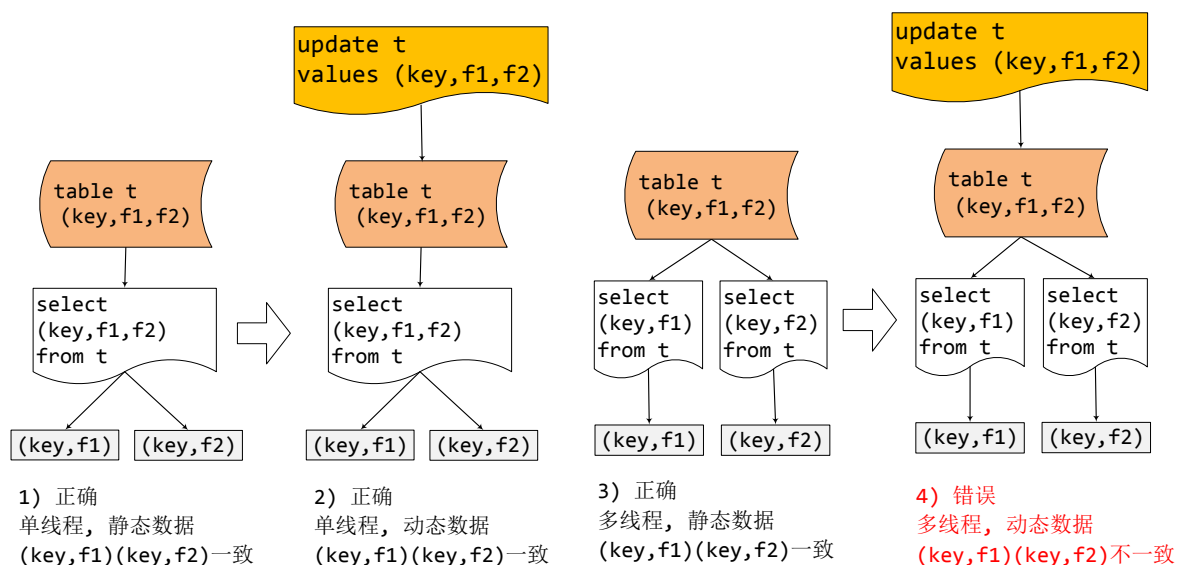


## 例 2

如下图，1 和 2 是当前线上运行的情况。在 1 中，我们能够保证读出来的(key, f1)和(key, f2)是同一条记录的数据，即二者是一致的；在 2 中，不管(key, f1)和(key, f2)是 update 之前还是 update 之后的数据，我们也能保证他们是一致的。

3 和 4 是为了提升性能而做的改造，将 select 操作分到两个脚本中执行。在 3 中，我们也能保证(key, f1)和(key, f2)的一致性；在 4 中，有可能出现(key, f1)是 update 之前的，而 (key, f2)是 update 之后的，即(key, f1)和(key, f2)不一致！

不幸的是，3 是线下测试的情况，4 是上线后的情况，线下测试没有问题，但是上线之后，由于数据不一致，导致回滚！



## 后果

数据错误

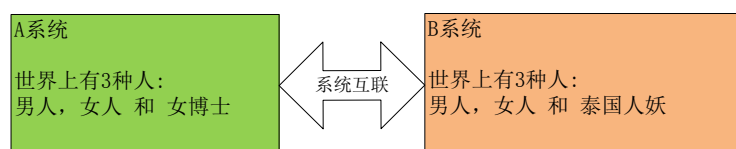
## 解决方案

☹

## 反模式23:忽视产品设计导致的一致性问题

### 例子

如图，女博士要哭了……



## 后果

影响用户体验

## 解决方案

设计评审考虑

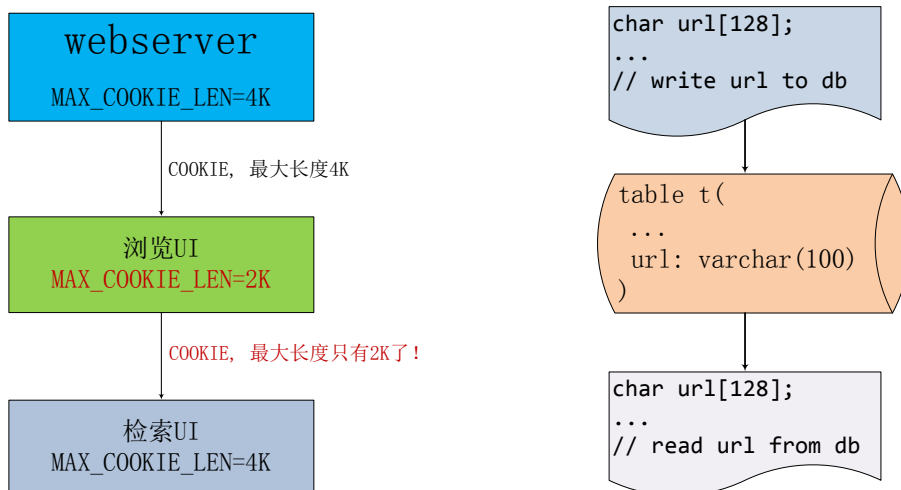
## 反模式24:忽视模块内部限制导致的一致性问题

### 例子

例：透传字段的截断

如左图，大部分浏览器对 **cookie** 的长度限制是 **4K**，但是浏览 UI 却把 **cookie** 的长度限制写成 **2K**，导致部分请求被截断。

如右图，数据库中对 **url** 字段的限制为 **100**，大量的 **url** 插入数据库后被截断。



### 后果

部分请求被截断，影响线上服务

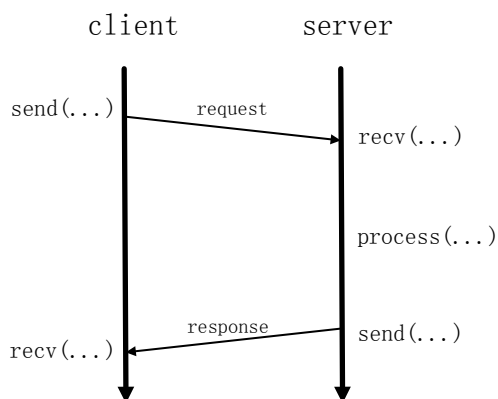
### 解决方案

对系统的一些限制需要整体规划，保持系统内部模块限制的一致性。

## 反模式25:忽视模块交互协议导致的一致性问题

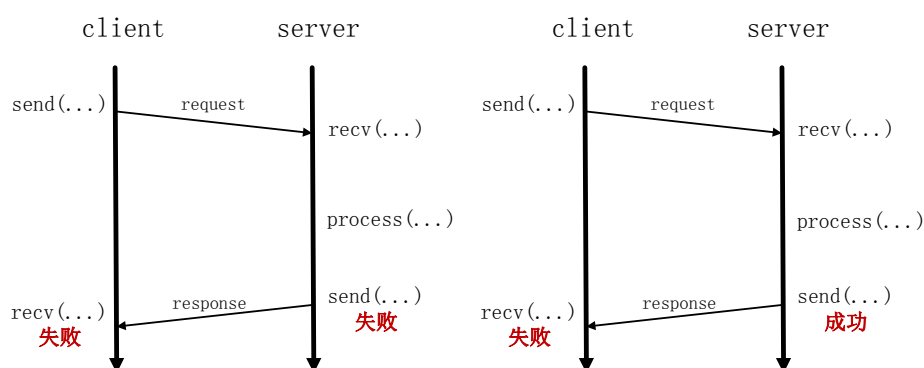
### 例子

一般上下游模块的交互协议如下





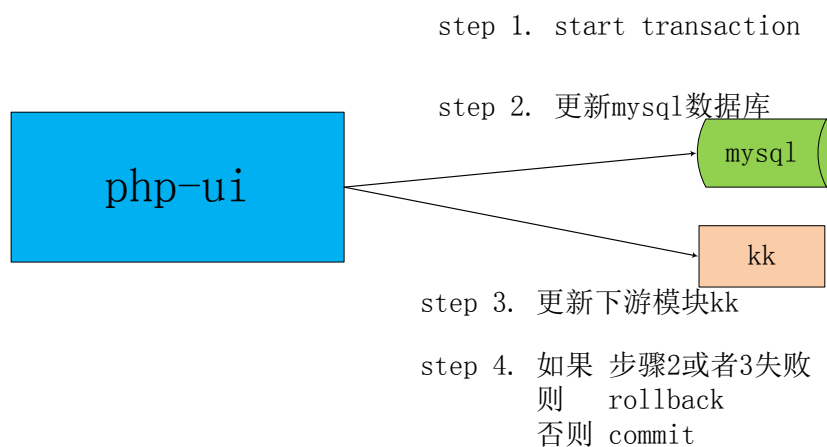
这个交互协议对于一致性要求很强的场合，是不适用的，考虑以下两种情况：



左图：**server send 失败，client recv 肯定失败**，这个是处理超时的场景，**server**已经处理好了，该写文件的已经写了，该修改数据库的也已经修改了，考虑回滚了吗？

右图：**server send 成功，client recv 失败**，这个是由于 **send** 返回成功只是保证数据已经拷贝到操作系统的内核缓冲区，并不能保证数据已经到达对方，发送数据的操作是在 **send** 返回之后，由操作系统进行的，如果由于网络故障，或者 **client** 模块挂掉，就有可能出现这种情况。这种场景下，**server** 根本无法知道数据是否已经到达 **client** 了。

如下，某系统中，对 **mysql** 和 **kk** 的数据，有很强的一致性要求，要么两者都更新，要么两者都不更新，所以把这两个事情放在一个事务中，在 **step 3** 中，只要出现上面左图和右图中的任意一个场景，**step 4** 就会回滚事务，从而出现 **mysql** 没有更新，**kk** 却已经更新的情况！



## 后果

系统中出现不一致的情况，对于强一致性的场合，这可能是无法接受的。

## 解决方案

如果一致性要求特别高，对极少数的数据不一致都无法容忍，那么可能需要设计能保证强一致性的交互协议；

如果一致性的要求不那么高，可以容忍少量的数据不一致，那么可以考虑一致性监控，做好处理不一致的预案。

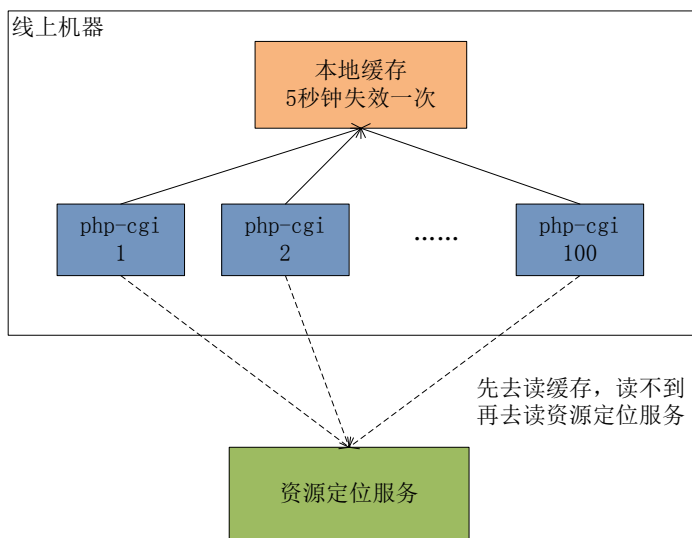
## ➤ 对外依赖

### 反模式26:爆发式的访问外部服务

#### 例子

如图，一台线上机器部署 100 个 `php-cgi` 进程，`php-cgi` 依靠资源定位服务获取后端模块的信息（地址，端口，超时等配置），为了提升效率，本地有一个公用的缓存，考虑到资源定位信息的时效性问题，缓存每 5 秒失效一次。

每当 5 秒这个时间点到达时，缓存失效，100 个 `php-cgi` 进程一窝蜂地去访问资源定位服务器，然后跑最快的那个 `php-cgi` 进程，即第 1 名选手获得资源信息，写入缓存中，接着第 2 名写缓存，第 3 名写缓存……直到第 100 名写缓存。接下来的 5 秒中，全部访问本地缓存，世界又恢复了平静。直到下一个 5 秒到达……



这种爆发式的访问，给资源定位服务带来极大的负担。在某一次资源定位服务的部分机器发生故障之后，这种爆发式的访问，成为压死骆驼的最后一根稻草。

这个像不像古老的系统中才存在的惊群问题？

#### 后果

降低服务稳定性

#### 解决方案

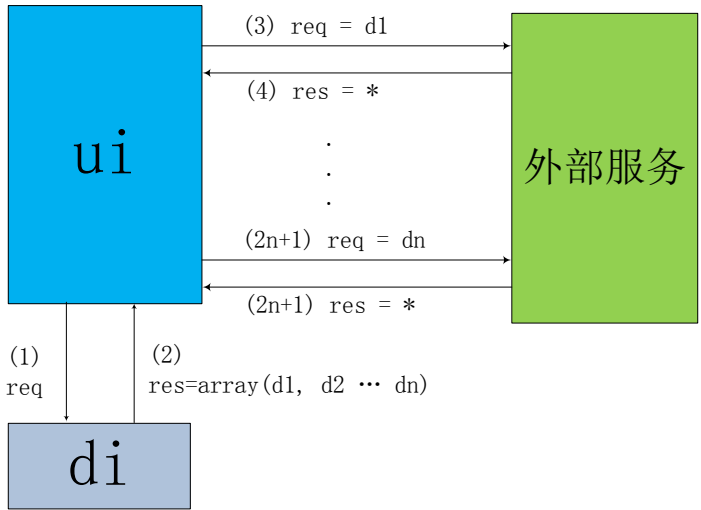
集中式访问，或者考虑本地缓存的平滑失效

### 反模式27:一个请求中不受限制地多次访问外部服务

#### 例子

例 1

如图，**ui** 首先从 **di** 获得结果列表(**d1**, **d2**, ... **dn**)，然后根据这个结果列表分别去请求外部服务，**ui** 与外部服务交互的次数取决于 **di** 返回的结果数 **n**。




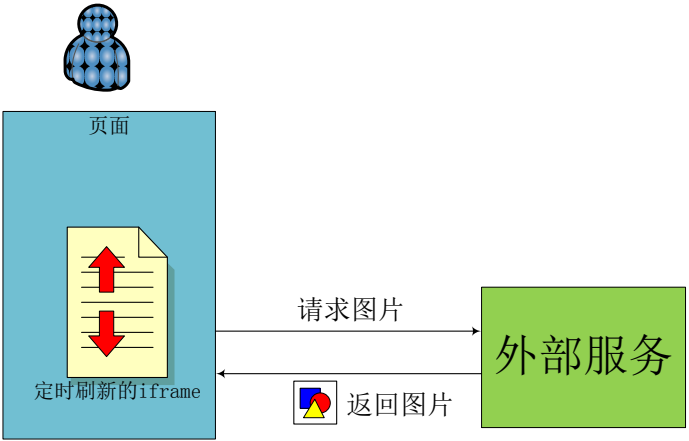
显然，这个访问模型有很大的风险，首先是 **ui** 与外部服务的交互时间不可控，与 **n** 相关，其次是外部服务要承受的压力是 **ui** 的 **n** 倍。

在现实场景中，这个 **n** 可能是由用户决定，代表用户提交的帖子数；**n** 也可能是由商户决定，代表商户提供的连锁店个数，优惠个数。在系统早期，**n** 比较小，问题没有暴露出来，随着系统逐渐膨胀，某些情况下 **n** 就突破了极限，线上就出现了大量的空白页，甚至是外部服务被压垮。

例 2

如图，某页面中嵌入一个定时刷新的 **iframe**，每次刷新都去外部服务那边请求一个图片。

你一定猜到了，这个故事是以外部服务被压垮，大量用户看到红叉做为结局（注：图片在网页上显示为红叉，像这样 ）。



后果

对外部服务压力大，系统稳定性风险高

解决方案

一般情况下，避免一个请求中多次访问外部服务，可以考虑批量处理，但是要注意数据量的限制。

如果一定要多次访问外部服务，那么要评估对外部服务的访问上限，并且做好监控。

## 反模式28:不以标准的方式使用外部服务

### 例子

某模块依赖于外部的一个服务，这个外部服务的返回结果是 json 格式。

一般情况下，我们都是以 json 库来解析返回结果，然后从中提取字段，但是这个模块却是以正则匹配的方式从 json 结果中提取字段，结果在外部服务的某次升级之后，字段顺序发生变化，导致这个模块功能异常，出现事故。

### 后果

易导致兼容性问题

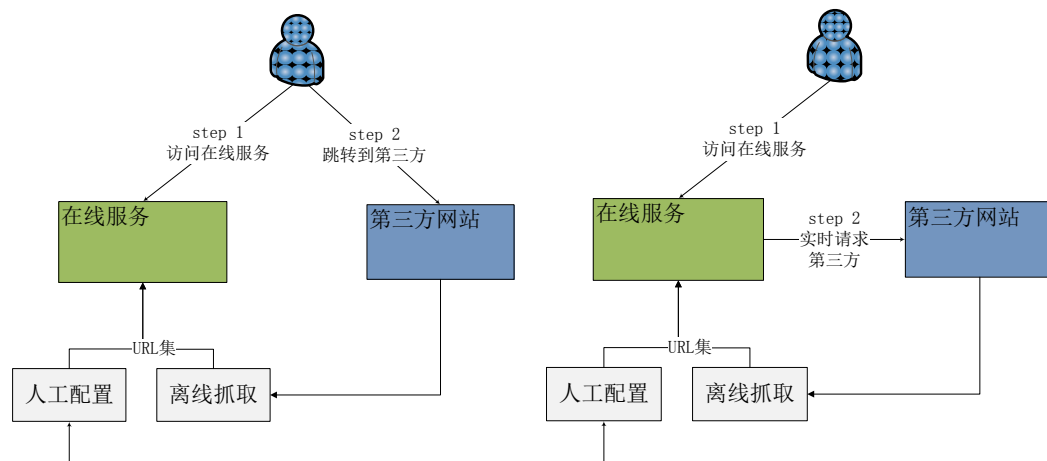
### 解决方案

严格遵守外部服务使用规范

## 反模式29:忽视外部服务的时效性

### 例子

如图，从离线抓取和人工配置到在线服务生效，需要有一定的时间间隔；两次离线抓取之间，也有一定的时间间隔。在这段时间间隔之内，第三方网站的资源可能已经发生了变化。



典型场景为：

- 1) 左图中 **step 2** 跳转的地址是死链或者是不相关链接
  - 2) 右图中，实时请求第三方的商品信息不存在，或者已经下架
- 这两种情况下，前者伤害用户体验，后者直接损失收入。

### 后果

伤害用户体验，或者损失收入

## 解决方案

评估在一定时间间隔内，与第三方内容不一致的比例，是否可以接受。

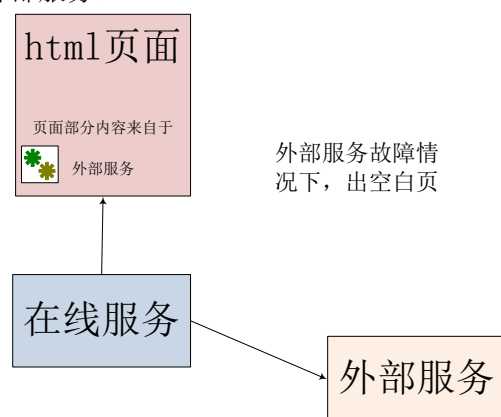
如果与营收相关，需要监控第三方网站情况（通过日志或者直接监控），第一时间获得变化通知

## 反模式30:外部服务故障时缺少对系统的保护

### 例子

#### 例 1

如图，某产品线，在线服务模块为用户生成一个 `html` 页面，这个页面的部分内容来自于外部服务。



但是这个系统没有做好外部服务故障情况下的自我保护，所以系统的稳定性很差，低于外部服务的稳定性，表现在：

1. 当外部服务无法服务时，系统没有做好容错处理，出空白页；
2. 当外部服务超时严重时，系统响应速度严重降低，甚至打不开；
3. 没有考虑外部服务的所有返回码，所以当外部服务返回新的返回码时，用户可能会看到莫名其妙的提示，而研发人员却完全不知道用户看到了什么。

### 后果

系统稳定性很差，低于外部服务的稳定性

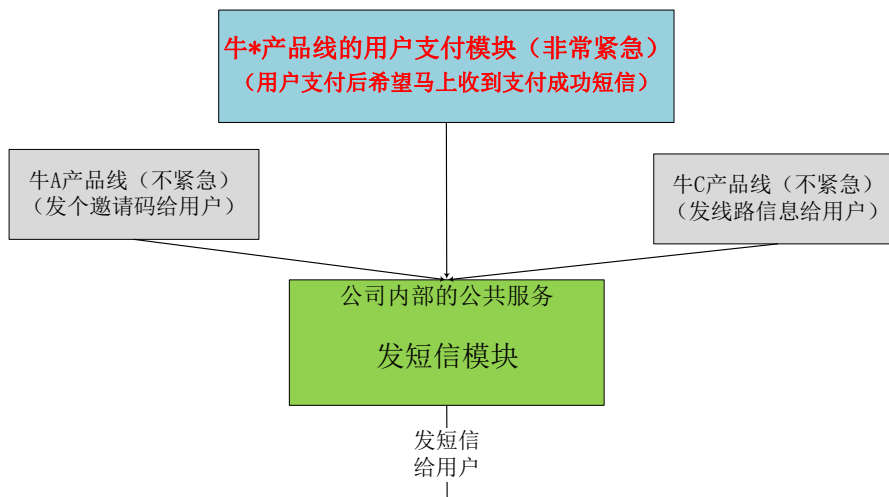
## 解决方案

考虑外部服务故障情况的系统自我保护，服务降级预案，报警机制

## 反模式31:不考虑外部服务的标准

### 例子

如图，一个叫做“牛\*”产品线（介于牛 A 和牛 C 产品线之间）在用户支付完成之后，需要依赖发短信模块向用户发送短信，这个操作是非常紧急的，稍有延迟都可能遭到用户投诉。



但是，发短信模块作为一个公共服务，不能保证短信能在多久时间之内发出去，所以，某一次在发短信模块出现大面积延迟之后，牛\*产品线收到用户大量投诉，而旁边的牛 A 和牛 C 产品线却没一点问题也没有。

## 后果

降低本产品线的服务等级

## 解决方案

在产品设计的时候，需要考虑外部服务不达标的情况（例如，添加一些提示，降低用户的焦虑，准备一些用户安抚措施等）

在架构设计时候考虑，评估外部服务的服务标准（如短信服务的最大延时，吞吐量等）

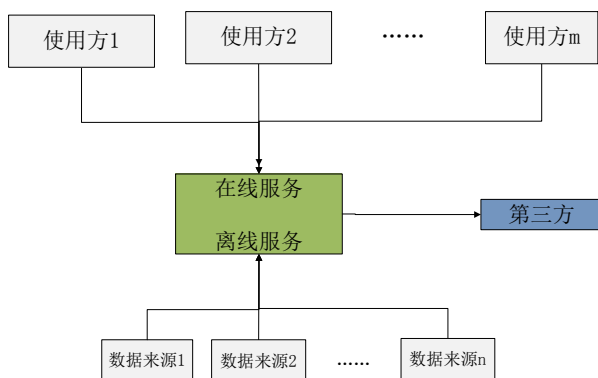
在运维上，考虑外部服务标准相关的监控，以及预案

## ➤ 对外服务

## 反模式32:把样例当接口规范

### 例子

某产品线的抽象模型如图，特点是接入来源多（入口），使用方也多（出口）。



该产品线没有对入口和接口做严格的规范约束，只有所谓的样例规范(拿例子当规范)，

或者口头规范，样例作为规范，很容易产生不同的理解，例如，线上曾经出现过

- 1) 类型理解不一致导致（一方认为 `int`，一方认为是 `string`）导致使用方大规模功能异常
- 2) 字段值为 `null` 导致使用方页面展现错误，甚至出现空白页
- 3) 使用方误用另外一个相似的字段，导致功能异常

```
{
  "content": {
    "detail_info": {
      // 类型是什么,int 还是 string
      "overall_rating": "3",

      // 下面这个字段，最大长度是什么，可否为 null?
      "poi_address": "长宁区仙霞路 602 号 2 楼(近水城路)",

      // 类似的两个字段,用哪个? 区别是什么
      "premium": null,
      "premium2": null,
      ...
    },
    ...
  },
}
```

随着入口和出口的增长，产品线的复杂度以  $m*n$  的趋势增长，经常需要与所有的使用方联调，也经常出现因为接口不明确导致使用方功能异常。

如果在初期将出口和入口用规范约束好，那么产品线的复杂度可以降低到  $m+n$ ，甚至是  $n$ 。

### 后果

三高：系统复杂度高，联调代价高，质量风险高

### 解决方案

维护实时更新的接口规范文档，并且严格遵守

## 反模式33:对外接口缺少请求来源字段

### 例子

### 后果

不利于追查问题和统计分析

### 解决方案

开放给多个产品线使用的接口应当把请求来源当作必选字段

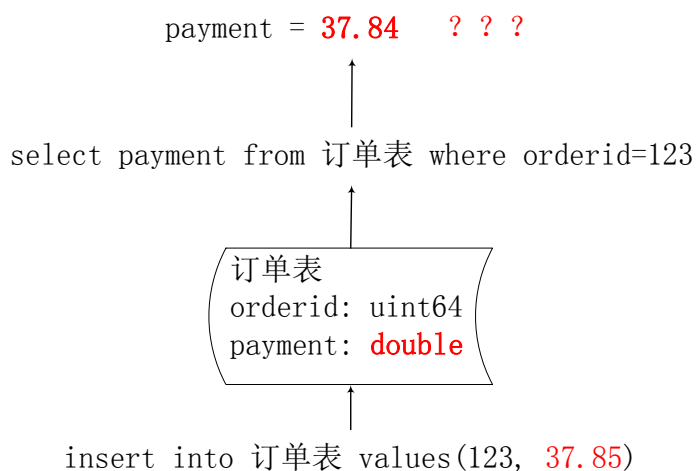
## ➤ 数据存储

### 反模式34:使用浮点数存储精确数值

#### 例子

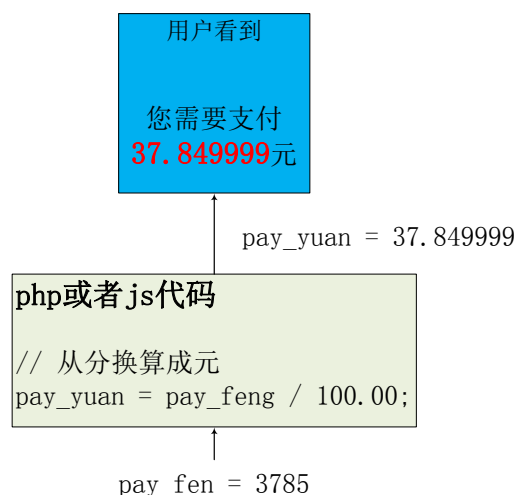
##### 例 1: 浮点数存储误差

由于浮点数表示的原因，在某些情况下，会出现 1 分钟的误差，但是对于一些敏感的数值，比如“付款金额”来说，这个误差是非常严重的！如下，存入 37.85 元，取出竟然变成了 37.84 元！



##### 例 2: 浮点数计算误差

从分转换到元时，使用了浮点计算，浮点计算出现误差。



这种情况，推荐的办法是先将 `pay_fen` 转换成字符串，然后做字符串操作，在倒数第二个数字之前插入一个小数点。

#### 后果



误差，在某些情况下，误差是致命的

### 解决方案

不允许使用浮点数存储，计算和传输精确数值。

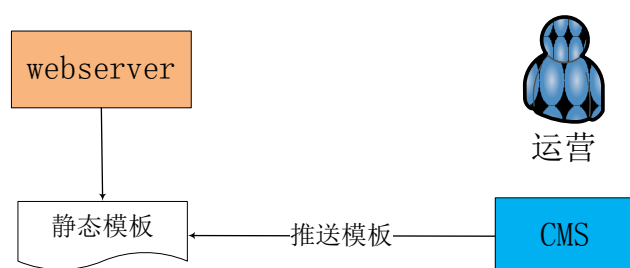
## ➤ 架构风险控制

### 反模式35:热加载风险

#### 例子

##### 例 1

如图，运营同学通过 CMS 系统随时向前端 **webserver** 推送静态模板（名人榜单，热销商品，今日话题等等），然后 **webserver** 利用静态模板，及时将最新信息展示给用户。



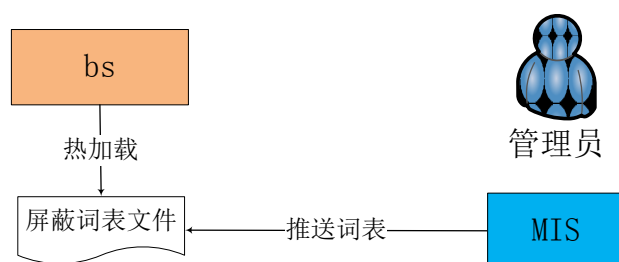
这个功能是很好的，但是往往因为对风险考虑不足引发事故，风险包括

- 1) 运营同学的误操作，或者 CMS 的 BUG，导致生成静态模板有问题，例如多个逗号，少个逗号，多个空格，少个空格，甚至生成一个空文件都有可能
- 2) 运营同学不是专业的技术人员，通过 CMS 推送静态模板后，不一定有能力，或者有意愿，对相关的功能做完整的回归。

某一次，运营同学不小心发布了一个空的文件，导致首页（就是首页）出现严重功能异常。从发生故障，到 OP 同学收到报警，迅速处理，到故障恢复，仅仅持续 3 分钟，但是 PV 已经损失了 100 万，造成严重事故。

##### 例 2

如图，管理员经常需要将更新的屏蔽词表文件（通过后台管理系统）推送到 **bs** 模块的目录下，然后 **bs** 模块热加载(reload)这个词表文件，从而实现屏蔽词功能的实时生效。



同例 1 一样，风险考虑不足容易引发事故，这些风险主要是两点：

- 1) 错误的词表引发 **bs** 功能异常：由于后台管理系统是离线系统，一般质量不会很高，再加上管理员容易有误操作，所以词表文件容易有这些问题

- a. 词表中某些词超长，超过了大家所认为的最大长度，比如 `MAX_WORD_LEN`
  - b. 词表超级大，超过的存放词表的数组大小，比如 `MAX_WORD_NUM`
  - c. 词表比正常情况小了很多，丢了很多数据
  - bs 如果对这些异常处理不周到，容易热加载失败，或者出 `core`，或者重新启动失败，或者功能异常。
- 2) 热加载相当于小的上线，但是团队却不一定处于上线状态：热加载的过程是随时进行的，这个时候整个研发团队很可能不在上班状态，如果出现异常，不能快速处理，一旦拖延，就会引发事故。

某一次，管理员配置了一条超级长的词，导致 `bs reload` 时候 `core dump`，因为后台管理系统是一下子把词表推送给所有 `bs` 的，所以一大片的 `bs` 同时 `core dump`，从而引发前端大面积超时，`OP` 同学迅速从食堂赶回来执行服务降级预案，从报警开始到服务恢复正常，经过了 40 分钟，损失流量 2000 多万，造成事故。

### 架构特征



### 风险点： 热加载失败或者热加载错误数据

#### ➤ 触发条件

由于人工误操作或者程序 `BUG`，导致生成的热加载数据错误；  
热加载程序校验不严，导致错误数据也被加载；  
热加载程序提供实时服务，导致出错后即使很快恢复，也会造成巨大影响

#### ➤ 风险后果

无法热加载；  
加载错误数据；  
影响在线服务

#### ➤ 解决方案

1. 用户输入的风险控制：`CMS` 和 `MIS` 系统对于人工输入，要做尽可能严格的检查，包括数据类型，格式，长度，字段之间互校验等；
2. 生成文件的风险控制：首先要保证输出文件格式正确性（可考虑格式校验工具），其次要保证文件内容正确（可考虑与上一个版本生成文件做 `DIFF`）；
3. 加载过程的风险控制：热加载模块需要对数据做严格的判断和异常处理，如果不能加载，需要继续使用旧的数据，并且打印报警；
4. 上线过程的风险控制
  - a) 系统可以提供热加载预览功能  
在运营提交操作之前，`CMS` 系统生成一份预览页面，明确告知点击提交之后的效果；
  - b) 系统可以提供 `DIFF` 和统计功能

在管理员提交操作之前，MIS 系统生成一份 DIFF 报告，明确告知本次提交的带来的影响（如增加多少数据，修改多少数据，输出多少数据），确认后再继续

- c) 上线过程可以考虑小流量，先小流量，回归一下功能，观察没有问题后，再继续上
- 5. 培训热加载相关的人员，如运营，热加载后需要去回归产品功能；
- 6. 回滚预案，热加载机器上需要有历史数据的备份，以便紧急情况下快速回滚；

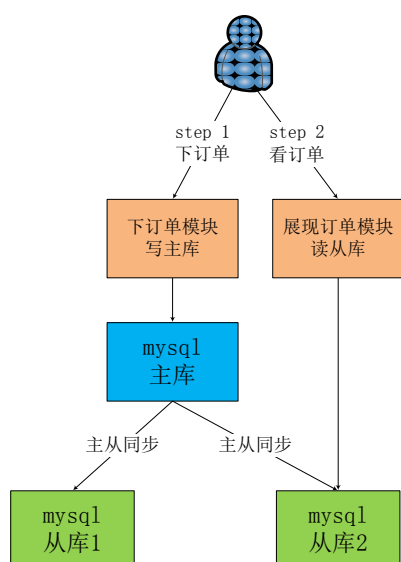
## 反模式36:数据库主从同步风险

### 例子

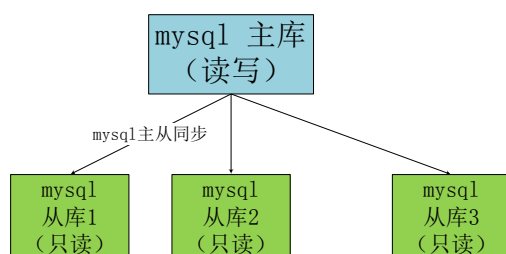
#### 例 1: mysql 主从同步延时

如下图，正常情况下，mysql 主从延时很短，用户下完订单之后跳转到个人中心能够看到订单，但是在真实的环境中，主从延时可能很长，比如因为网络抖动，主库突然有大量更新，或者某些从库机器负载高等原因，主从延时会达到 500 秒以上，这个时候，会出现大量用户下完订单，却看不到订单的情形。

另外，不同从库的同步延时可能还不一样，这会导致展现结果不稳定。



### 架构特征



### 风险点 1: 主从延时不可控

#### ➤ 触发条件

网络抖动；主库大量读写；慢查询；主库操作加锁(某些 SQL 语句会隐含加锁)

➤ **风险后果**

用户看到不一致的数据；数据错误

➤ **预防措施**

设计上考虑主从延时要求；慢查询优化；分库；对延迟敏感的业务强制读主库；避免主库大量读写；主从同步监控

**风险点 2：从库之间不一致**

➤ **触发条件**

根据网络状况，机器负载情况，从库之间的不一致持续时间会加大

➤ **风险后果**

用户看到不一致的数据；数据错误

➤ **预防措施**

设计上考虑从库之间不一致的容忍极限；主从同步监控

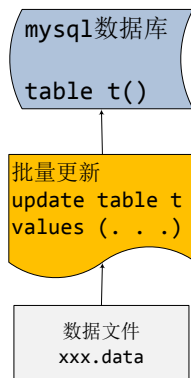
## 反模式37:数据库批量更新风险

### 例子

如下图，某产品线经常需要批量更新数据库，数据来源为公司外部的合作方。

某一次，合作方系统出现故障，导入的数据大面积出错，等到用户发现的时候，数据已经导入完成了，由于缺少应急预案，不得不临时编写修复脚本，然后启动修复。从发现问题，到修复完成，已经过去 10 几个小时了。

### 架构特征



**风险点 1：来源数据质量不可控，导致数据库大面积错误**

➤ **触发条件**

来源数据方的程序 bug，人工操作失误，或者一些不可控因素

➤ **风险后果**

➤ **解决方案**

建立数据准入机制：如更新之前，先生成数据质量报告，人工审核报告+数据抽查后，才允许进行批量更新。

## 风险点 2：脚本容错处理不到位，导致丢失数据

- 触发条件  
数据库交互超时；个别数据不合法；
- 风险后果  
随机丢失数据
- 解决方案  
做好脚本的容错处理：个别数据更新不成功，可以考虑重试或者其它恢复策略，恢复无效的，需要记录，然后由人工介入处理。

## 2. 编码反模式

### ➤ 变量初始化

### 反模式38:新加变量不考虑初始化

#### 例子

如下是典型的服务器模块代码。其中 `thread_data_t *td` 用来保存线程数据。

```
// 线程数据，每个线程一份
typedef struct {
    ...
    int a;
    string b;
    map<int, string> c;
} thread_data_t;

// 初始化线程数据
void reset_thread_data(thread_data_t *td)
{
    ...
    td->a = 0;
    td->b = "";
    td->c.clear();
}

int process(thread_data_t *td)
{
    // 使用 td 中的变量 td->a, b, c
}

// 处理一个请求
```

```

int server_callback(thread_data_t *td)
{
    ...
    reset_thread_data(td);
    read_request(td);
    process(td);
    send_response(td);
    ...
}

```

某一次升级，在线程数据中添加一个变量 d

```

typedef struct {
    ...
    int a;
    string b;
    map<int, string> c;

    vector<string> d;
} thread_data_t;

```

// reset\_thread\_data 函数没有修改

**// 悲剧了，忘记初始化 d 了**

```

void reset_thread_data(thread_data_t *td)
{
    ...
    td->a = 0;
    td->b = "";
    td->c.clear();
}

```

```

int process(thread_data_t *td)
{

```

// 使用 td 中的变量 td->a, b, c

**// 勇敢地使用 td->d**

td->d.push\_back(...); **// BUG: d 中包含上一次请求处理留下的数据**

```

}

```

## 后果

模块出现不可预期行为

## 解决方案

新加变量，必须考虑初始化，请检查所有引用这个变量的地方，确保都经过初始化

## 反模式39:更改变量作用域不考虑初始化

### 例子

如下是典型的服务器模块代码。其中 `thread_data_t *td` 用来保存线程数据。

```
// 线程数据，每个线程一份
typedef struct {
    ...
    int a;
    string b;
    map<int, string> c;
} thread_data_t;

void reset_thread_data(thread_data_t *td)
{
    ...
    td->a = 0;
    td->b = "";
    td->c.clear();
}

int process(thread_data_t *td)
{
    ...
    vector<string> d;

    // 狠狠地使用 d
    d.push_back(...);
}

// 处理一个请求
int server_callback(thread_data_t *td)
{
    ...
    reset_thread_data(td);
    read_request(td);
    process(td);
    send_response(td);
    ...
}
```

某一次升级，需要将变量 `d` 的作用域从函数级提升到线程级

```
typedef struct {
```

```

...
int a;
string b;
map<int, string> c;

    vector<string> d;
} thread_data_t;

// reset_thread_data 函数没有修改
// 悲剧了，忘记初始化 d 了
void reset_thread_data(thread_data_t *td)
{
    ...
    td->a = 0;
    td->b = "";
    td->c.clear();
}

int process(thread_data_t *td)
{
    ...

    // 勇敢地使用 td->d
    td->d.push_back(...); // BUG: d 中包含上一次请求处理留下的数据
}

```

## 后果

模块出现不可预期行为

## 解决方案

更改变量作用域，必须考虑初始化，请检查所有引用这个变量的地方，确保都经过初始化

## ➤ 函数接口

### 反模式40:函数接口包含 trick

#### 例子

这是 public/mcpack 基础库中的代码。

```

// 成功: 返回 mc_pack_t*, 合法的指针
// 失败: 返回(mc_pack_t*)errno, 其中 -255 < errno < 0, 代表错误号
// 这里使用了 trick

```



```

// 因为  $-255 < \text{errno} < 0$ , 所以  $(\text{mc\_pack\_t}^*)\text{errno}$  不会与普通指针混淆
mc_pack_t * mc_pack_open_r(const char * buf, int size, ...);
const void * mc_pack_get_raw(const mc_pack_t * ppack, ...);

// 一般人用法, 指针跟 NULL 比较
ptr = mc_pack_open_r(buf, size, ...);
if (ptr != NULL)
    // do something
    // BUG: may be core dump!

// 正确用法
ptr = mc_pack_open_r(buf, size, ...);
if (MC_PACK_PTR_ERR(ptr) == 0) // 区分 ptr 是合法指针还是错误号
    // ok, do some thing.

```

## 后果

引诱犯罪, 几乎所有新 rd 都会犯这个错

## 解决方案

接口设计考虑易用性, 拒绝 trick

```

// 如下, 返回值代表 errno, 参数 ptr 存放返回的指针, 各司其职, 互不相干
int mc_pack_open_r(const char * buf, int size, ..., mc_pack_t **ptr);
int mc_pack_get_raw(const mc_pack_t * ppack, ..., mc_pack_t **ptr);

// 正确用法 1
ret = mc_pack_open_r(buf, size, ..., &ptr);
if (ptr != NULL)
    // do something

// 正确用法 2
ret = mc_pack_open_r(buf, size, ..., &ptr);
if (ret == 0)
    // do something, ptr must be ok

```

## 反模式41:函数接口升级不考虑返回值的兼容性

### 例子

#### 例 1

某基础库, 版本 v1.0 代码如下

```

/**
 * @brief ...
 * @param ...

```

```

* @return int 返回错误码
* @retval 成功时返回 0
*          失败时返回对应的错误码，错误码<0
**/
int galileo_query_resource(...);

```

升级到版本 v2.0 后，代码如下

```

/**
 * @brief ...
 * @param ...
 * @return int 返回错误码
 * @retval 成功时返回 0
 *          成功时返回 1 -- 这是版本 v2 新加的一种成功方式
 *          失败时返回对应的错误码，错误码<0
 **/
int galileo_query_resource(...);

```

在版本 v1.0 中，rd 这么写代码，这是完全正确的

```

if (galileo_query_resource(...) != 0) {
    // 错误处理
}

```

而升级到版本 2.0 后，上述代码将另外一种成功方式返回的 1 当作错误来处理，导致在某些情况下，模块重启失败。这个重启失败是某一次重大事故的主要原因。

## 后果

让旧代码失效，引发线上问题

## 解决方案

接口升级严格保证兼容性：包括对参数，返回值，行为的考虑；

例子中，要么在版本 v1.0 的时候就明确指出<0 代表失败，>=0 代表成功；要么在版本 v2.0 时候，新增加一个函数 galileo\_query\_resource\_ext，重新指定返回值的含义。

## 反模式42:函数接口升级不考虑参数的兼容性

### 例子

如图，这是一段排序的代码，根据 sort\_type 对 items 数组排序。左边为重构之前的代码，右边为重构之后的代码。

<pre> int sort_items(vector&lt;item_t*&gt; &amp;items, int sort_type) {     bool is_descend_order = false; // 是否降序      switch (sort_type) {         case 0:             is_descend_order = true;             break;         case 1:             is_descend_order = false;             break;         case 2:             is_descend_order = true;             break;         case 3:             is_descend_order = false;             break;         default:             is_descend_order = false;             break;     }      item_compare_t comparer(is_descend_order);      sort(items.begin(), items.end(), comparer);      // ... } </pre>	<pre> int sort_items(vector&lt;item_t*&gt; &amp;items, int sort_type) {      item_compare_t comparer(sort_type);      sort(items.begin(), items.end(), comparer);      // ... } </pre>
<pre> class item_compare_t { public:     item_compare_t(bool is_descend) {         m_is_descend = is_descend;     }      bool operator()(const item_t *a, const item_t *b)     {         if (m_is_descend)             ...         else             ...     }  private:     bool m_is_descend; }; </pre>	<pre> class item_compare_t { public:     item_compare_t(int sort_type) {         switch (sort_type) {             case 0:                 m_is_descend = true;                 break;             case 1:                 m_is_descend = false;                 break;             case 2:                 m_is_descend = true;                 break;             case 3:                 m_is_descend = false;                 break;             default:                 m_is_descend = false;                 break;         }     }      bool operator()(const item_t *a, const item_t *b)     {         if (m_is_descend)             ...         else             ...     }  private:     bool m_is_descend; }; </pre>

由于程序中有多次类似 `switch` 语句块的代码，所以根据重构法则，将这个 `switch` 语句块提取出来，移到 `item_compare_t` 的构造函数中，相应的，构造函数的原型由

`item_compare_t(bool is_descend)`

变成

`item_compare_t(int sort_type)`

应该说，这是一个很好的重构，如果没有以下的线上问题的话。

重构过程中，遗漏了某一个 `switch` 语句块，这个语句块还是以 `bool` 的方式调用构造函数，即：

```
item_compare_t comparer(is_descend)
```

因为 `bool` 类型可以安全地提升为 `int` 类型，所以编译能够通过。遗漏的这个 `switch` 语句块是比较不容易走到的分支，所以测试也没有回归到，上线后，用户报告部分功能异常，接到用户反馈后，本次上线立即回滚。

## 后果

让旧代码失效，引发线上问题

## 解决方案

函数参数个数变化，类型变化，修改默认参数，需要检查所有调用者。

# 反模式43:可写缓冲区当作参数传递却不附带缓冲区长度

## 例子

### 例 1

标准 C 库中，所有“**参数中不带目标缓冲区长度**”的字符串函数，例如

- ✓ `strcpy`  
`char *strcpy(char *dest, const char *src);`  
不传目标缓冲区 `dest` 长度
- ✓ `strcat/strncat`  
`char *strcat(char *dest, const char *src);`  
`char *strncat(char *dest, const char *src, size_t n);`  
两个函数都不传目标缓冲区 `dest` 长度，其中第二个函数的参数 `n` 是最多拷贝字节数，不是 `dest` 或者 `src` 的长度。
- ✓ `scanf("%s", buff)/gets(buff)/sprintf(buff, fmt, ...)`  
不传目标缓冲区 `buff` 长度

历史上，这些臭名昭著的函数，造就了无数的 `core dump` 和缓冲区溢出攻击。

### 例 2

如下函数原型含有致命缺陷：如何保证每一个调用者所传递的 `buff` 都是足够大的？随着调用次数增多，这个溢出风险就变得不可控了。

```
int CacheUnit::hit_cache(const uint *sign, void *buff)
{
    if (not hit cache)
        return 0;

    char * data = ...;
```

```

    int    size = ...;

    memcpy(buff, data, size); // BUG: 缓冲区溢出!

    return size;
}

```

更合理的函数原型是

```
int CacheUnit::hit_cache(const uint *sign, void *buff, int size);
```

## 后果

缓冲区溢出，程序出 core

## 解决方案

可写缓冲区当作参数传递，必带缓冲区长度

数组当作参数传递，必带数组长度

## ➤ 异常处理

### 反模式44:不检查其它模块传来的数据

#### 例子

某模块读请求的代码，不对读出来的数据做检查，导致模块很脆弱，上下游模块随便发一个错误的数据包过来，都可能导致模块 core dump。

```

// 读请求
int ret = nshead_read(sockfd, req_head,
    req_buff, sizeof(req_buff), ...);

if (ret != NSHEAD_RET_SUCCESS) { /* 错误处理 ...*/ }

// 打开 mcpack
mc_pack_t *pack = mc_pack_open_r(req_buff,
    sizeof(req_buff), // BUG: 脏数据风险, 第二个参数
    // 应为 req_head->body_len
    ...);

if (MC_PACK_PTR_ERR(pack)) { /*错误处理 ... */ }

// BUG: 没检查返回值, qt 和 num 可能是脏数据
mc_pack_get_int32(pack, "qt", &qt);
mc_pack_get_int32(pack, "num", &num);

```

```
// ...

// BUG:不检查 qt 范围, qt 越界会导致 core dump
... = command_table[qt];

// BUG:不检查 num 范围, 可能会导致内存撑爆
char *inner_buff = malloc(num * sizeof(some_struct_t));
```

## 后果

模块稳定性低, 系统健壮性差, 一个模块出 bug, 会引起系统内一大片的模块异常。

## 解决方案

读入其它模块的数据之后, 马上对数据做尽可能严格的检查, 包括接口版本, 数据包大小, 字段是否存在, 字段类型, 取值范围等。

## 反模式45:不注意检查函数返回值

### 例子

#### 例 1

```
short_item *newitem = make_space(item->content_size());

// BUG: 应检查 newitem
if(item == 0)
    return MC_PE_NO_SPACE;

// newitem==NULL 时,core dump
memcpy(newitem, item, item->size());
```

#### 例 2

```
ret = fscanf(input, "%s\t%s\t%d\t%d\t",
             contsign, objurl, &height, &width);

if (ret <= 0)          // BUG! 应该与 4 比较
    // ...
```

#### 例 3

```
ret = snprintf(output, sizeof(output),
               "%s\t%s\t%d\t%d\t", contsign, objurl, height, width);

if (ret < 0)
    // ...

// use output
```

```
// BUG: snprintf 的手册上说的明白
//      在缓冲区 output 不够大的时候, 返回 > sizeof(output) 的值
//      所以, 这里 output 的值可能是非法的
```

#### 例 4

库函数中, mmap 函数原型为:

```
void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

这个非常自然的写法, 却是错误的:

```
ptr = mmap(...);
if (ptr == NULL)    // BUG! 应该与 MAP_FAILED(-1) 比较
```

#### 例 5

这是一段 shell 脚本

```
cmd1 | cmd2 | cmd3
if [ $? = 0 ] ...    # BUG! 应该判断 $PIPESTATUS 数组
```

### 后果

程序稳定性差, BUG 多多

### 解决方案

认真检查函数的返回值:

- 1) 如果是库函数, 需要查阅手册;
- 2) 如果是自己或者别人写的代码, 需要查阅注释, 甚至需要看一下函数代码, 弄明白一共有哪些返回值。

## 反模式46:不做异常处理

### 例子

例 1: 这是一段脚本:

```
$path = ....

mkdir -p $path
cd $path
rm -rf *

# do xxx
# do yyy
```

一旦拼接出来的 \$path 不合法, 或者 mkdir 失败, 后面的 rm 操作直接把当前目录下的所有文件全删除了, 连这段脚本自己也被删除了, OMG, 自己把自己删除了!

例 2: 又是一段脚本, 建库脚本里面摘出来的  
为什么举脚本的例子, 因为脚本最容易忽略异常处理。

```
# 生成 input.txt
```

```
# 以下这行简单的代码会给我们什么样的惊喜呢?
```

```
sort input.txt > output.txt
```

```
# 使用 output.txt
```

`sort` 是 `shell` 命令, 一般人不会考虑失败的情况。但是在临时文件空间不足(因为可能需要外排序), 或者临时文件没有写权限, 或者内存不足, 等等, 这些情况下, `sort` 真的是可以失败的, 如果失败, 那么 `output.txt` 的内容是未定义的。

某一次, 因为 `/tmp` 空间满了, 所以 `sort` 失败了, 建库脚本建出来的脏数据, 通过传库程序传到线上机器后, 大量用户就看到了许多不该看到的页面, 造成事故。

如果我们在 `code review` 的时候指出 `sort` 这行要加异常处理, 很多同学会争辩: “这个怎么可能失败, 线上有机器监控, 磁盘监控呢。”

需不需要做异常处理, 这个例子已经给了回答。

## 后果

程序稳定性差, BUG 多多

## 解决方案

每一步都认真做好异常处理

- 1) 如果是调用基础库, 需要认真查阅手册, 搞清楚有哪些异常类型, 分别处理
- 2) 如果是调用自己或者别人的函数, 需要查阅注释, 甚至需要看一下函数代码, 弄明白一共有哪些异常情况, 分别处理
- 3) 如果是 `shell` 命令, 需要查阅 `man` 手册, 一般都需要做异常处理。如果是与文件或者网络相关, 那就毫无疑问, 必须做。

## 反模式47:拷贝缓冲区不考虑长度

### 例子

例 1

不管 `strlen(name.c_str())` 和 `sizeof(buff)` 的大小关系如何, 以下代码都有可能  
导致程序 core dump:

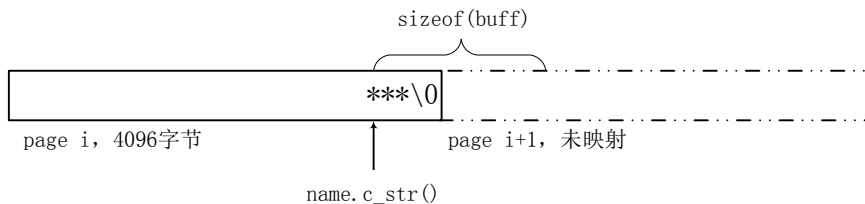
```
// std::string name;  
// char buff[BUFF_SIZE];  
// ...
```

```
memcpy(buff, sizeof(buff), name.c_str());
```



当 `strlen(name.c_str()) >= sizeof(buff)` 时, `buff` 不以 `\0` 结尾, 后续读 `buff` 会越界;

当 `strlen(name.c_str()) < sizeof(buff)` 时, `memcpy` 还是会拷贝 `sizeof(buff)` 字节。如果 `name.c_str()` 刚好位于操作系统 `page i` 的末尾, 那么从 `name.c_str()` 开始拷贝 `sizeof(buff)` 字节会越过 `page i` 到 `page i+1`, 而如果 `page i+1` 刚好没有被映射, 那么 `memcpy` 就会访问非法内存, 然后 `segment fault`, `core dump`。



### 后果

`segment fault`, `core dump`

### 解决方案

所有缓冲区操作, 必须严格检查来源和目标的缓冲区长度

## ➤ 基础库使用

### 反模式48:重复实现基础库已经实现的代码

#### 例子

靠, 这个不是二分查找吗? 基础库已经实现的代码, 为什么重新实现一遍? 浪费时间不说, 而且很容易有 `BUG`, 后续别人也不好维护。

别小看二分查杂算法, 要写对很难的: 第一个二分查找算法早在 1946 年就被发表, 但第一个没有 `bug` 的二分查找算法却是在 12 年后才被发表出来!

```
template<class Type>
int BinarySearch(Type *data, int low, int high, Type key,
                 int (*pfunc)(Type&, Type&) = DefaultFunc)
{
    // ...
}
```

### 后果

浪费时间, 容易出 `BUG`, 维护代价高

### 解决方案

多学习, 多使用基础库

## 反模式49:忽视第三方库的线程安全性

### 例子

某模块线上 core dump，导致出 core 的代码如下。

```
KvsDataServ* KvsDataServ::get_instance(pthread_t index) {  
    if (_map_instance.count(index) == 0) {  
        // BUG: 线程不安全  
        _map_instance[index] = new KvsDataServ();  
    }  
    return (KvsDataServ*) _map_instance[index];  
}
```

`std::map _map_instance` 是全局的，虽然每个线程的 `index` 不同，但是，多个线程同时对 `std::map` 做 `insert` 操作，容易写坏数据，导致出 core。

### 后果

程序容易出 core

### 解决方案

多线程场景下，慎重选用基础库；考虑引入库的线程安全性

## ➤ 基础库开发

## 反模式50:共享库中使用全局变量

### 例子

### 后果

### 解决方案

## ➤ 性能相关

## 反模式51:加锁区间内做长耗时操作

### 例子

例 1：加锁区间内做网络操作，这是百度 `ub_client` 基础库曾经出过的 BUG

```
// from ub_client.cpp
```

```

int ub_fetchsock(svr_t *svr)
{
    // ...

    pthread_mutex_lock(&svr->mutex);

    // ...

    // BUG: 在 svr->mutex 这个大锁下, 如果某一次 connect 偶然超时,
    // 那么 1 秒内, 该 svr 上的所有 connect 操作都要在这个锁上等待

    new_sock = ub_tcpconnecto_ms(
        svr->ip, svr->port,
        svr->connect_timeout /*默认 1 秒*/);

    // ...

    pthread_mutex_unlock(&svr->mutex);

    // ...
}

```

## 例 2: 加锁区间隐含文件系统操作

如下是访问 cache 的代码, 为了实现可持久化, cache 是采用 mmap 文件映射内存技术实现的。

```

int read_from_cache(uint32_t key[2], char* buff, int buff_size)
{
    // ...

    pthread_mutex_lock(&chche_mutex);

    // ...

    if (hit_the_cache(key, &hit_addr, &hit_size)) {
        // BUG
        // hit_addr 指向文件映射的内存(mmap)
        // 它对应的内存可能还在磁盘上, 还没有读入内存,
        // 这个时候, 线程挂起, 操作系统去读磁盘,
        // 读写磁盘操作, 特别是随机读, 很可能导致长耗时
        memcpy(buff, hit_addr, hit_size);
    }
}

```

```

        // ...

        pthread_mutex_unlock(&cache_mutex);

        // ...
    }

```

### 后果

性能下降，长耗时请求增多，甚至死锁

### 解决方案

需要特别检查加锁区间内的所有的长耗时操作：如网络操作，文件系统操作等。

## 反模式52:随意清零大块内存

### 例子

某线上模块的性能比较低，只能达到 120q/s，而且 cpu 打满，将 `memset` 清除改成标记清除之后，性能提升 10 倍，达到 1300q/s。

```

#define QL_THREAD_BUFFER_SIZE (100*1024*100)

// 接收请求

// 性能问题: memset 10M 内存!
memset(data_buffer, 0, QL_THREAD_BUFFER_SIZE);

// 处理请求

```

### 后果

降低性能

### 解决方案

避免 `memset` 大块内存，一般情况下，不要 `memset` 超过 1M 的内存。

## ➤ 编码规范

## 反模式53:直接使用 `reserved` 字段

### 例子

ui 和后端 as 模块交互，as 返回的是一个结构体，定义为

```

typedef struct {
    ...

```

```
        unsigned int    reserved:16; // 保留
    } qdispinfo_t;
```

某一次升级，需要新加一个字段，记录用户积分，**直接使用 reserved**，ui 和 as 模块重新编译，于是两个模块中隐藏了很多直接使用 reserved 字段的肮脏代码：

```
qdispinfo_t* info = ...;

info->reserved = ...;

... = info->reserved;
```

一个月之后的另外一次升级，另外一个同学想使用 reserved 字段，将结构体改成这样，

```
typedef struct {
    ...
    unsigned int    reserved:11; // 保留
    unsigned int    rank_level:5; // 使用 reserved 字段里面的 5 个 bit
} qdispinfo_t;
```

只修改 ui，不修改 as，ui 重新编译，于是 ui 模块中就有了这样的代码

```
qdispinfo_t* info = ...;

info->reserved = ...;

... = info->reserved;

info->rank_level = ...;
```

上线后，悲剧了，本来是 16 个 bit 的用户积分信息，高 5 个 bit 直接被修改了，所有积分大于 2047(写成二进制是 11 个 1)的用户积分全部错误。

## 后果

代码不可维护，随时引发错误

## 解决方案

尽量不使用 reserved 字段；

如果非要使用，请改名后再使用（第一个升级），

修改结构体后，确保所有使用该结构体的模块都重新编译，一起上线。

## 反模式54:手工计算代替编译器做计算

### 例子

### 例 1

在一大堆代码中，要保证这两个 128 同时修改，这样的代码为将来升级埋下了隐患：

```
char name[128];

// ...

snprintf(name, 128, ...);
```

建议修改为：

```
snprintf(name, sizeof(name), ...);
```

### 例 2

老代码，手工算的， $10 \times 61440 + 102400 = 716800$ ，一切 OK！

```
#define MAX_ALADING_NUM    10      // 最多 10 个 alading 结果
#define MAX_ALADING_LEN    61440   // 每个 alading 结果最大 60K
// ...
#define MAX_RESULT_BUFFER  102400  // 正常结果大小 100K

// ...
// 这个文件中另外一个地方的定义，离前面很远很远
#define MAX_US_BUFFER      716800  // 10 个 alading 结果+正常结果
```

几个月以后的一次升级，需要将单个 alading 结果的最大大小从 60K 调整为 100K：

```
#define MAX_ALADING_NUM    10      // 最多 10 个 alading 结果
#define MAX_ALADING_LEN    102400  // 每个 alading 结果最大 100K
// ...
#define MAX_RESULT_BUFFER  102400  // 正常结果大小 100K

// ...
// 这个文件中另外一个地方的定义，离前面很远很远
#define MAX_US_BUFFER      716800  // 10 个 alading 结果+正常结果
```

悲剧了， $10 \times 102400 + 102400 > 716800$ ，在某些情况下，缓冲区太小导致结果被截断，线上用户看到了错误的页面。

建议修改为：

```
#define MAX_US_BUFFER      \
    ( (MAX_ALADING_NUM)*(MAX_ALADING_LEN) + (MAX_RESULT_BUFFER) )
```

### 后果

代码难以维护，为后人挖坑，升级时候容易触发 BUG

### 解决方案

涉及 size, offset 相关的数值，尽量不要手工计算，而要采用编译器计算。

需要同步修改的数值，在代码中写在一起，并加上显眼的注释。  
代码中避免直接写死数字。

### 3. 运维反模式

#### ➤ 上线过程

#### 反模式55:上线时用 cp 更新 bin 和 so 文件

##### 例子

某产品线上线时，需要更新 php-cgi 的一个扩展，上线命令如下

```
$ cp new_dir/libmcpack.so lib/libmcpack.so
$ bin/php-fpm restart
```

结果导致机器上 100 个正在使用这个扩展的 php-cgi 进程几乎在同时出现 core dump，一个 php-cgi 进程的 core 文件大小是 2.4G，那么 100 个就是 240G，这是一个可怕的数据量，按照磁盘写带宽 180M/s 计算，大约需要 22 分钟，这段时间内机器 cpu 打满，idle 直接降为 0，停止响应请求，root 用户都无法登录机器。

令事情更加严重的是，所有前端机器同时都出现了这种情况，在长达 30 分钟的时间内，该产品线完全停止对外服务，构成重大事故。

对于正在运行的 bin 和 so 文件，正确的更新方式是 mv 而不是 cp

```
$ mv new_dir/libmcpack.so lib/libmcpack.so
$ bin/php-fpm restart
```

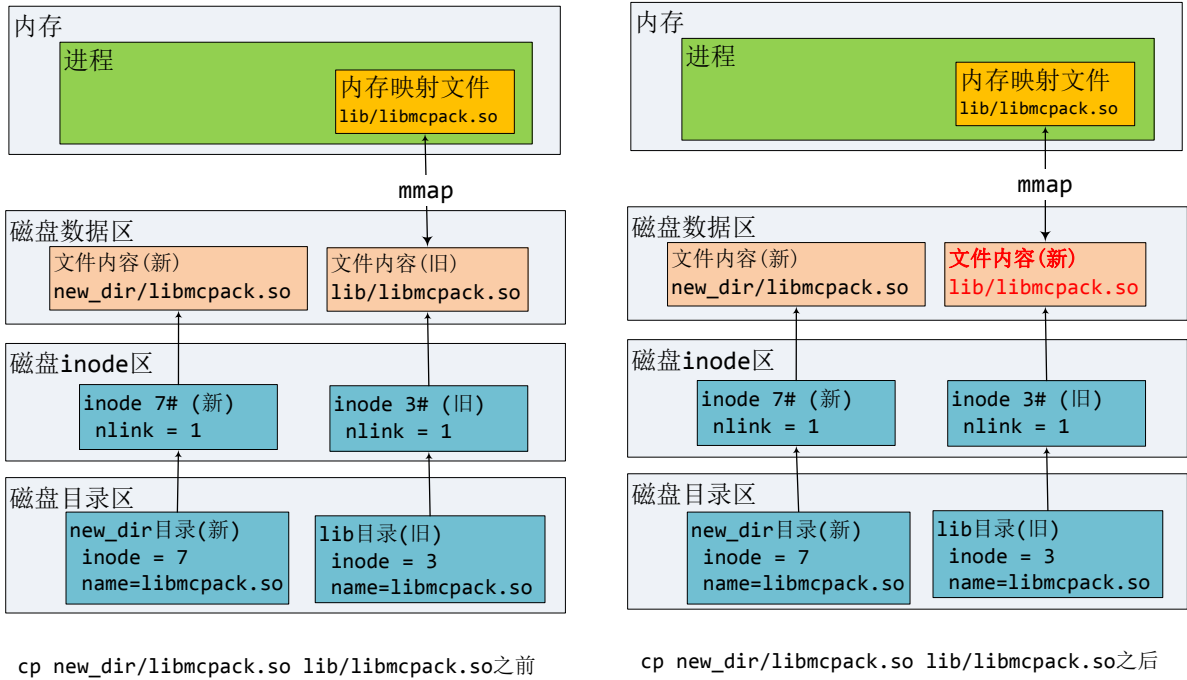
很多同学疑惑，为什么用 cp 有问题而用 mv 就没问题了呢？我们用 Linux 系统自带的 strace 工具来对比一下 cp 和 mv 命令的执行过程。以下例子假设源文件和目标文件都已经存在，并且在同一个文件系统上。

```
$ strace cp new_dir/libmcpack.so lib/libmcpack.so
...
open("new_dir/libmcpack.so", O_RDONLY) = 3
open("lib/libmcpack.so", O_WRONLY|O_TRUNC, 0100644) = 4
read(3, "...", 4096) = 4096
write(4, "...", 4096) = 4096
...

$ strace mv new_dir/libmcpack.so lib/libmcpack.so
...
rename("new_dir/libmcpack.so", "lib/libmcpack.so") = 0
...
```

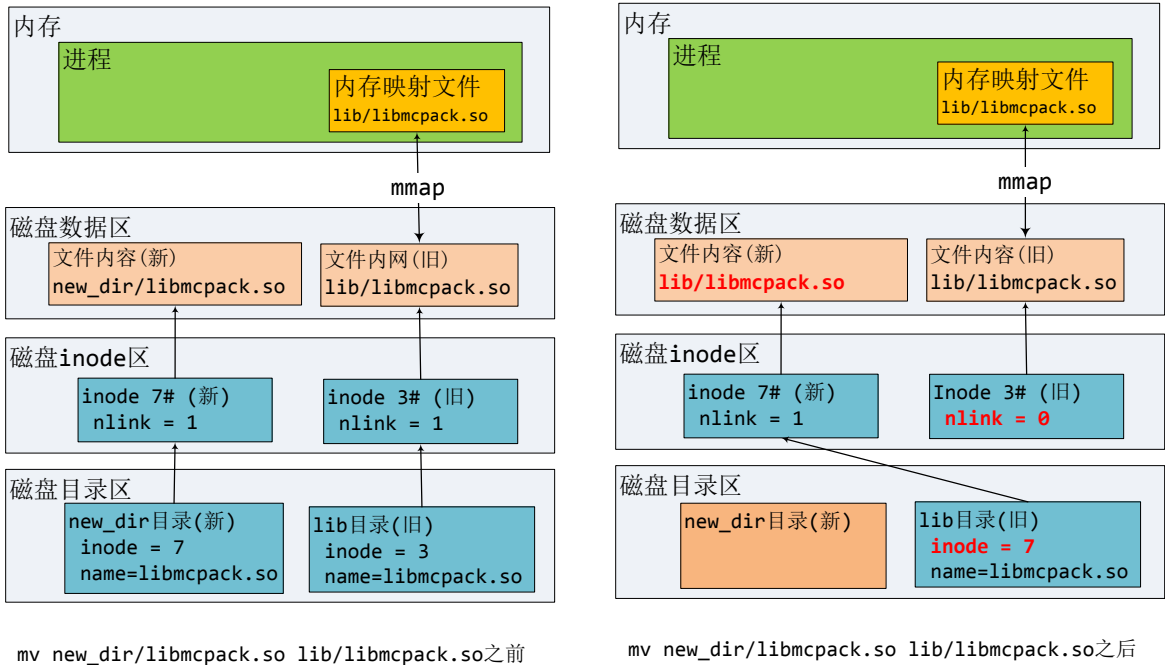
可以看到，cp 的时候，会首先将目标文件截断为 0(O\_TRUNC)，这个时候，如果目标

文件正在被访问（以 `mmap` 映射方式），那么就有可能出现访问越界，或者访问到错误的数  
据。一般情况下，程序会收到 `SIGBUG` 信号，然后 `core dump`。



与 `cp` 方式会修改文件内容不同，`mv` 方式不修改文件内容，只修改目录项（见下图）

1. 修改旧目录 `lib` 中的文件名和 `inode` 号
2. 删除新目录 `new_dir` 中的文件名和 `inode` 号



如果 `mv` 之前旧文件正在被访问，只要程序不重启或者重新 `open` 文件，那么 `mv` 之后程  
序还是访问的是旧文件。

由于 `mv` 修改了目录，所以，`mv` 之后，通过目录 `lib/libmcpack.so` 访问的是新文件  
了，而通过目录 `new_dir/libmcpack.so` 则找不到文件。



## 后果

使正在运行的程序出现 core dump

## 解决方案

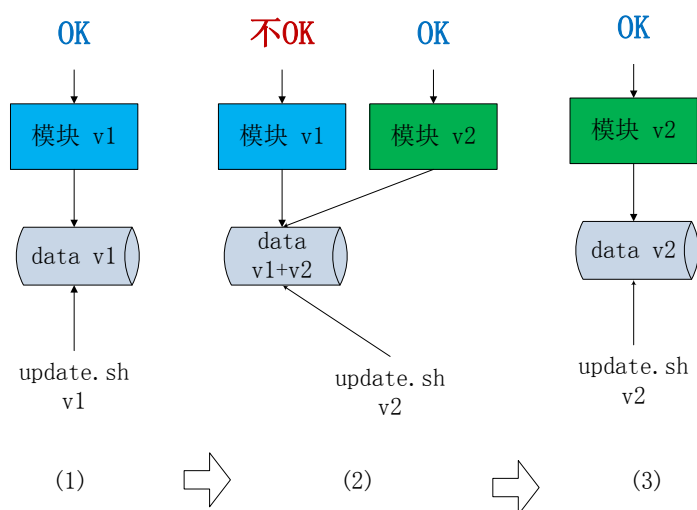
用 mv 代替 cp 命令更新 bin 和 so 文件

## 反模式56:忽视上线过程中的系统一致性问题

### 例子

例 1: 数据与模块的一致性

如图，上线前和上线后的状态分别是状态(1)和(3)，这两个是 OK 的，(2)是上线过程中的状态，这个时候模块 v1 服务是不正常的，由于各种原因，状态(2)停留的时候可能会比较长，影响线上服务，造成回滚。



## 后果

上线时影响线上服务，造成回滚

## 解决方案

上线过程中的**中间状态**也需要经过严格评估和测试

## 反模式57:忽视架构的一致性要求

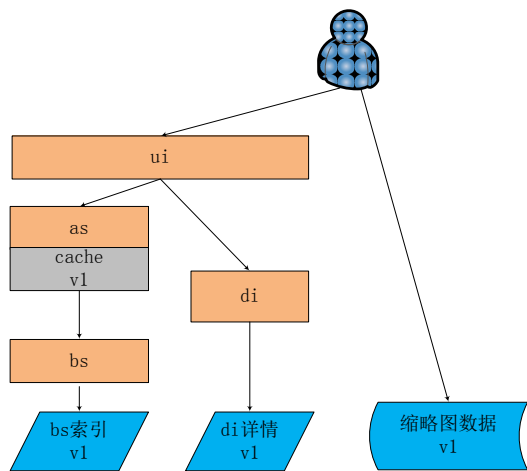
### 例子

这是某产品线曲折的上线故事。

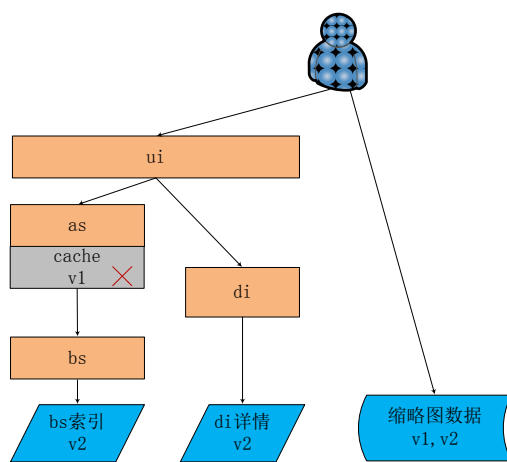
产品线背景是：bs 索引用于检索，di 存放被检索出来的信息详情，缩略图数据存放被检索出来的信息的缩略图，这三者需要严格一致，否则就会出现展示信息错误。

1. 图(1) 一切 OK
2. 图(2) 上线新版本数据 v2，但是忘记清 cache 了，导致命中 cache 的请求返回结果与检索词不相关，影响当日 3%的 PV

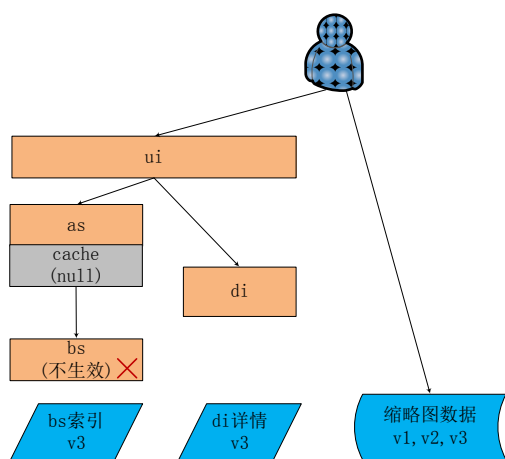
- 图(3) 上线新版本数据 v3，但是因为一些原因，bs 无法正确加载 v3 索引，上线失败，所以 v3 不生效，影响当日 5.4%的 PV
- 图(4) 由于某种原因，只能回滚到 v1，但是由于系统设计原因，v1 版本的缩略图数据刚好过期被删除，所以，前端用户看到的部分信息没有缩略图。



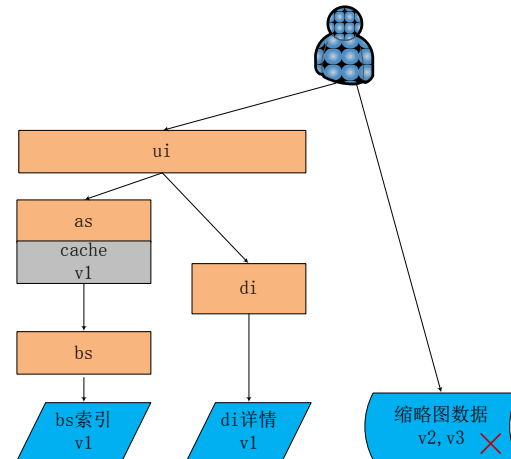
图(1) OK



图(2) 上线 v2，但是忘记清 cache



图(3) v3 上线失败，不生效



图(4) 缩略图 v1 过期

## 后果

上线过程中，影响线上服务，损失流量，如果处理时间过长，易造成事故

## 解决方案

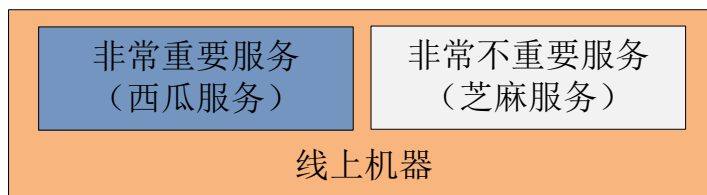
运维上重视架构的一致性要求

## ➤ 线上环境

## 反模式58:同机部署不同重要性的服务

### 例子

某产品线一台线上机器上同时部署了 **cache** 服务（非常重要服务）及另一个数据库服务，供产品线内部人员后台查询是用（非常不重要服务），在每天的流量高峰期，CPU 和 IO 的压力都比较大，产品线人员如果在这个时候执行后台查询服务，就会导致 **cache** 服务大量超时。



### 后果

不重要的服务影响重要服务

### 解决方案

考虑将重要服务与不重要服务分离部署

## 反模式59:没有防攻击预案

### 例子

这是某产品线的线上问题记录：

- 1) 3月8日，**lighttted** 两次受到攻击，压力暴涨
- 2) 3月9日上午，RD 检查线上 **lighttted** 配置，发现**无防攻击配置**
- 3) 3月9日晚上，RD 和 QA 经过实验，修改线上 **lighttted** 配置，加入防攻击配置
- 4) 3月10日上午，**lighttted** 再次遭受攻击，证明防攻击配置**没有生效**；RD 分析判断通过升级 **lighttted** 程序、动态库来解决问题
- 5) 3月10日晚上，升级 **lighttted** 上线，上线一台观察，发现新 **lighttted** 防攻击配置**没有生效**，每秒约损失约××个 PV，所以回滚
- 6) 无奈之下，OP 只好手动将攻击的 IP 加入黑名单，暂时减少损失
- 7) 事后统计，PV 总共损失××万

### 后果

发生攻击时，不能快速止损，甚至还可能手忙脚乱加重损失

### 解决方案

制定防攻击预案，并经过演练

考虑服务分级，必要时通过牺牲非核心服务，或者服务降级来减少损失

## 反模式60:监听端口号大于 1 万

### 例子

线上 **as** 模块，监听端口号是 **17000**，重启时候，经常出现 **bind()** 报告无法绑定 **17000** 端口，导致重启失败，有时候还会 **core dump**，日志中提示信息如下：

“bind: Address already in use”

检查代码，发现 bind 之前已经设置了 SO\_REUSEADDR 选项，为什么还会出现端口已被占用的情况呢？

那是因为线上机器不仅存在许多 server socket，还存在 client socket，这两类 socket 都是需要一个端口号的，server socket 端口一般是配置文件中指定的，而 client socket 端口，一般是在调用 connect() 时由系统选择，选择范围由 /proc/sys/net/ipv4/ip\_local\_port\_range 指定，我们的机器上一般是 [10000, 61000]，如下

```
$ cat /proc/sys/net/ipv4/ip_local_port_range
10000 61000
```

如果在 as 被 kill 之后，重启之前，17000 端口恰好被某个 client socket 的 connect() 选中，as 再 bind 这个端口时，就出现 “Address already in use” 了。

我们不可能要求系统中所有的 client socket 都设置 SO\_REUSEADDR 选项，所以，我们只好让 server socket 避开这个端口选择区间。

## 后果

增加运维代价，上线时候导致报警，刺激大家本来就很脆弱的神经

## 解决方案

监听端口号改成小于 1 万

## 反模式61:重要数据无权限控制

### 例子

线上 memcache 无权限控制，qa 用线上配置做测试，向线上 memcache 写了大量垃圾数据，使大量用户看到了这些不该看到的数据。

某产品线 qa 向线上发送删除广告命令，导致线上广告一条条消失，通报批评，罚款。

## 后果

线下误操作情况下，容易写坏线上数据，影响线上服务

## 解决方案

线上重要数据加权限控制，线上环境和线下环境隔离

## 反模式62:重要数据无备份

### 例子

某产品线的 /home/xxx/yyy/mysql 目录，保存产品线重要的统计数据，为产品线提供决策依据。该数据至关重要，但是没有定期备份，这份数据全宇宙唯一。

在某一次磁盘清理过程中，在/home/xxx/yyy目录下，意外地执行了 `rm -rf *` 命令，1 分钟后发现错误，立刻停止了/home 分区的写入，但是为时已晚，大量数据已被删除。由于没有备份，只好采取文件系统的数据恢复方式，但是只能恢复部分数据。

## 后果

“万一”情况下，后果很严重

## 解决方案

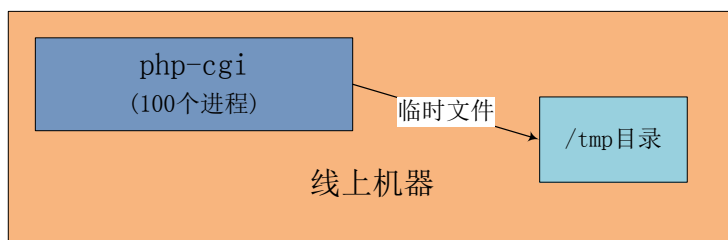
话说不怕一万，就怕万一，其实在线上环境中，这些万一是很平常的事，磁盘坏，机器坏，链路断，误操作，宇宙射线等等，所以要是不能接受万一的后果，那就坚持重要数据定期备份。

## 反模式63:不清理过期数据

### 例子

#### 例 1

如图，线上 php 程序接收用户上传的文件，处理过程中，需要在 tmp 目录下建一个临时文件。由于这些临时文件没有定期清除，日积月累，所以 tmp 目录中的文件越来越多。

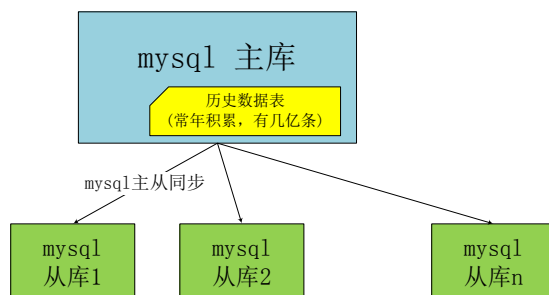


系统执行 `open` 或者 `stat` 系统调用的时候，需要对目录文件做线性扫描，时间复杂度是  $O(n)$ ，其中  $n$  为该目录下的文件个数。

终于在某一天的流量高峰，由于在 tmp 目录执行 `open` 和 `stat` 的时间太长，导致处理用户请求出现大规模超时，造成 PV 损失 600 多万。

#### 例 2

如图，mysql 数据库中存在一个历史数据表，经过多年的积累，这个数据表达到几亿的数据量。某一天需要删除这个表里面大约 1 亿的历史数据。突然删除这么多的数据给数据库带来巨大的压力，由于未制定好严密的方案，导致数据库主从延迟最大达到 5 个小时，造成事故。



## 后果

造成机器磁盘满，IO 性能下降，影响线上服务，易造成事故。

另外，线上程序大部分都没有判断 `write` 或者 `fwrite` 的返回值，所以，磁盘满之后，可能会出现大规模的数据错误，问题会很严重，厂长会很生气

## 解决方案

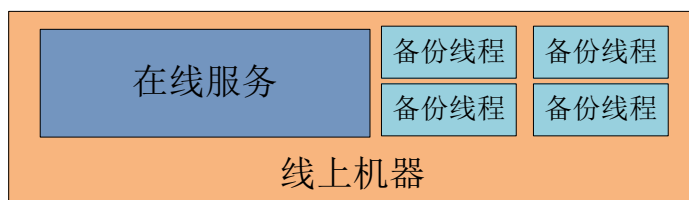
定期清理过期数据，包括日志文件，临时文件，数据库中一些记录，不要等到垃圾堵住家门口了再开始清理。

## 反模式64:线上机器执行大 IO 类任务

### 例子

#### 例 1

某产品线在线上机器上执行备份的系统多任务并行下载，IO 负载过大，多台机器假死，造成 PV 损失 2.4 万。



## 后果

影响线上服务

## 解决方案

控制线上机器 IO 任务的负载

## 反模式65:冗余度不足

### 例子

#### 例 1: 机器冗余度不足

如图，线上 bs 索引比较大，分成 210 层，根据不同层的索引的重要性，分成大库（从 1 到 30 层，每层部署 4 列）和小库（从 31 到 210 层，每层部署 1 列），其中不同层之间的数据不同，相同层之间不同列的数据相同。

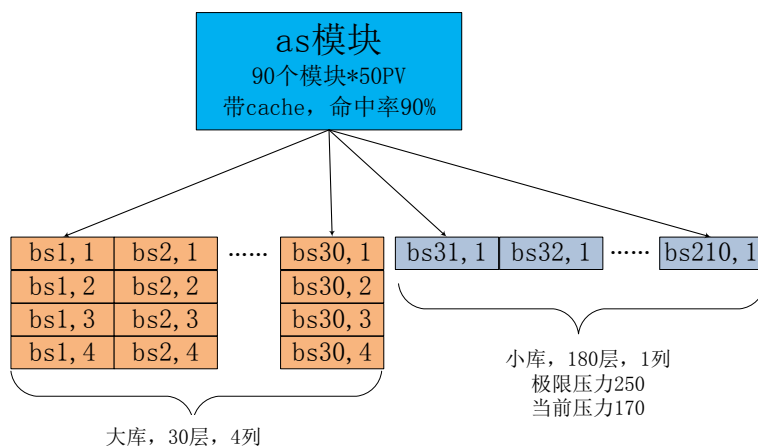
as 发起检索时候，一般是同时向所有层的 bs 发送请求，如果同一层有多列，只向其中的一列发请求。

现在我们考虑一下 as 的冗余度问题，假设 90 个 as 中，有一个 as 出现故障（死机或者重启），那么从 31 层到 210 层的小库 bs 增加的压力最多是 50（因为一个 as 的 PV 是 50，看图），如果两个 as 故障，小库 bs 增加的压力最多是 100，这已经超过小库 bs 的极限压力了（ $170+100>250$ ）。

从以上简单计算可知，as 的冗余度不到 2，也就是说 90 个 as 中，不能允许 2 个 as 同时出现故障。这个限制一方面对 as 的稳定性提出极高的要求，另一方面，也要求 as 上线和回滚时候要一个一个重启，拖慢了上线和回滚速度。

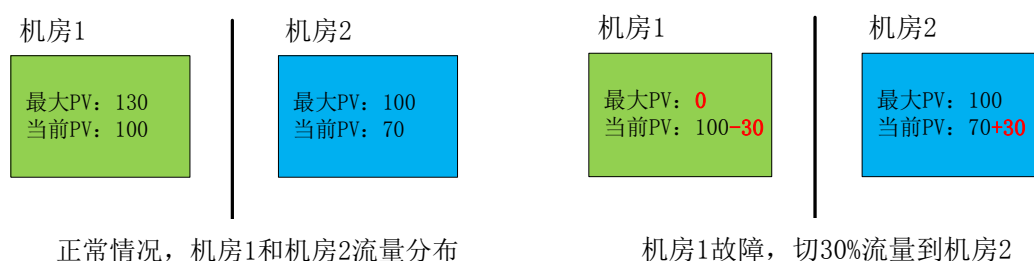
这个冗余度问题造成了多起事故。

(注：为简单起见，以上的分析省略了 as 的重查穿透机制)



## 例 2：机房冗余度不足

如图，某一次机房 1 出现服务故障，需要紧急把流量切到机房 2，但是由于机房 2 不能承受机房 1 的全部流量，所以只能看着机房 1 还剩余的 70%流量白白损失掉。



## 后果

稳定性容易出问题，造成事故

## 解决方案

评估机器冗余度，机房冗余度，还有所依赖的第三方的冗余度（其它产品线的，或者公司外的），保证冗余度符合产品线的稳定性要求

## ➤ 线上监控

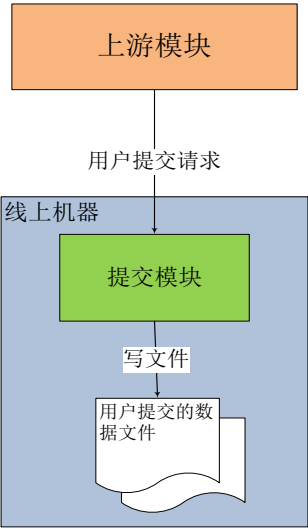
## 反模式66:缺少机器监控

### 例子

#### 例 1：缺少磁盘监控

某产品线的提交模块所在机器缺少磁盘监控，某一天，突然磁盘满，提交模块写文件失败，OP 同学接到上游模块报警后，跑回来紧急处理，手忙脚乱之间又添了点乱子，加重了损失。事后统计，损失用户提交请求 32 万，占全天提交请求总数的 4.5%，造成事故。

如果该机器有磁盘监控，在磁盘快满的时候报警，我们有充足的时间处理，这个事故是可以避免的。



**后果**

不能在故障之前尽早发现问题，以便尽早处理，容易造成事故。

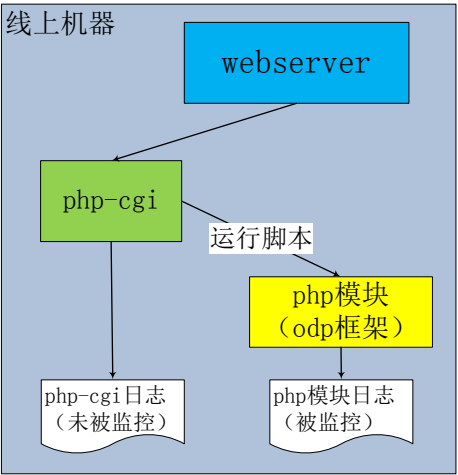
**解决方案**

线上机器（特别是新上线机器，紧急扩容机器），需要确保有完整的机器监控，如磁盘监控，内存监控，CPU 监控，网络流量监控等。

**反模式67:不监控模块日志**

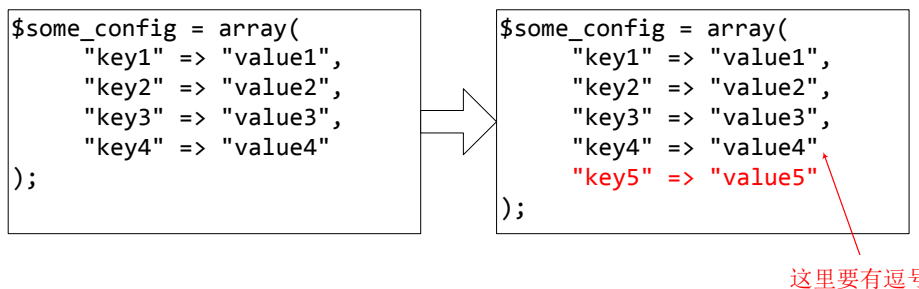
**例子**

例 1：不监控模块日志  
如图为典型的 php 模块部署：



线上监控了 php 模块的日志，但是没有监控 php-cgi 的日志。某一次小升级，修改了 php 模块中的一个 php 文件，但是不小心漏掉了一个逗号：





在某些情况下，**php-cgi** 打印 **fatal** 日志，然后终止处理请求。但是因为没有 **php-cgi** 日志监控，**php** 模块日志也没有异常，所以这个问题没有很快被发现，造成流量损失。

## 后果

不能尽快发现问题，容易造成事故

## 解决方案

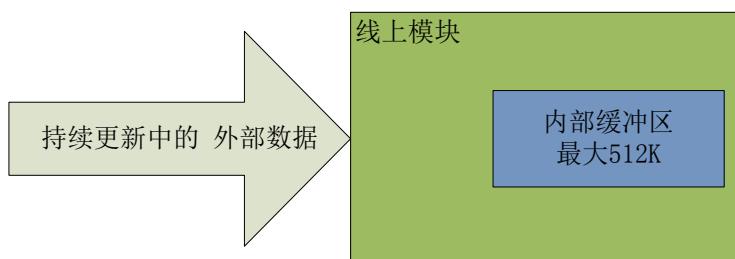
线上模块需要有完整的日志监控，除了监控 **fatal**，**warning** 外，还可以通过日志监控模块的平均处理时间和压力值，如果出现大的变动需要报警。

## 反模式68:不监控模块的内部限制

### 例子

如图，线上模块处理外部数据，外部数据是持续更新中的，模块内部有个 **512K** 的缓冲区，这个 **512K** 可能是头文件中定义的常量，也可能是配置文件中的一个配置项。

模块运行初期，**512K** 的内部缓冲区足够大，一切功能正常。但是随着外部数据的持续膨胀，终于有一天突破了内部缓冲区的限制，导致模块功能异常。



线上曾经出现过 **bs** 模块因为索引数据量增大，导致换库失败，造成 **PV** 损失；**di** 模块因为数据量增大，导致写回给 **ui** 模块的数据量超过内部缓冲区大小，导致结果被截断，前端出现空白页面。

## 后果

模块的内部限制在不知不觉中被突破之后，模块功能异常，容易造成流量损失

## 解决方案

- 1) **limit** 监控预防问题发生:即定时将模块内部限制的最大值和当前值(**MAX** 和 **CUR**)写到 **limit** 文件中去,然后监控这个 **limit** 文件,一旦 **CUR/MAX** 超过一定的阈值,例如 **90%**,就报警,然后人工介入处理。

- 2) 日志监控保证问题尽早被发现并且被处理：程序中，对于超过模块限制的请求，打印出清晰的 **FATAL** 日志，并且有日志监控程序报告出来。
- 3) 维护系统全局的限制，进而推导出模块的内部限制，避免不同模块的限制条件互相冲突。

## 反模式69:不监控产品线的核心指标

### 例子

某检索产品线提供商品信息的检索服务，某一次上线，由于配置文件错误，导致商品信息减少一半，几天后才发现，构成重大事故，如果有商品信息总量的监控，这个事故是可以避免的。

某产品线提供商户信息的检索服务，经常由于系统内部 **BUG** 导致某些商户无法展示，从而导致商户投诉，而如果有商户信息能否展示的监控，很多投诉是可以避免的。

### 后果

线上出问题后不能尽快发现，将小问题放大成重大事故

### 解决方案

梳理出产品线的所有核心指标，并做好监控，定时发送监控报告，如果核心指标异常，立刻发送报警